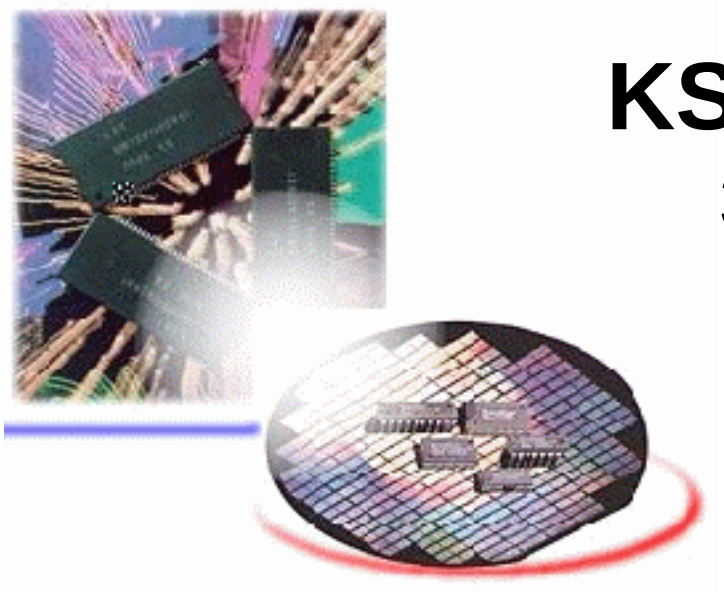
Excellence in Low-Power

The way MICOM/DSP should be



KS32C5000(A)/KS32C50100

32-bit RISC Microcontroller for Network Solution

Mar. 1999

SAMSUNG ELECTRONICS

www.DataSheet4U.com

Contents

- Network Protocol
  - What is Network ?
  - OSI Reference Model and TCP/IP
  - TCP/IP Networking Software & Basic Protocol
  - Real-time Operating System
- Real-time Operating System
  - Developping System with pSOSystem
  - pSOSystem BSP
  - Developping System with Nucleus
  - Nucleus H/W Device Driver
- Applicable System with SAMSUNG's NetMCU
  - Managed HUB
  - Managed Switching HUB
  - Router / Layer-3 Switching
  - Printer Server
  - Network Printer
  - Cable Modem
  - UPS Management Controller

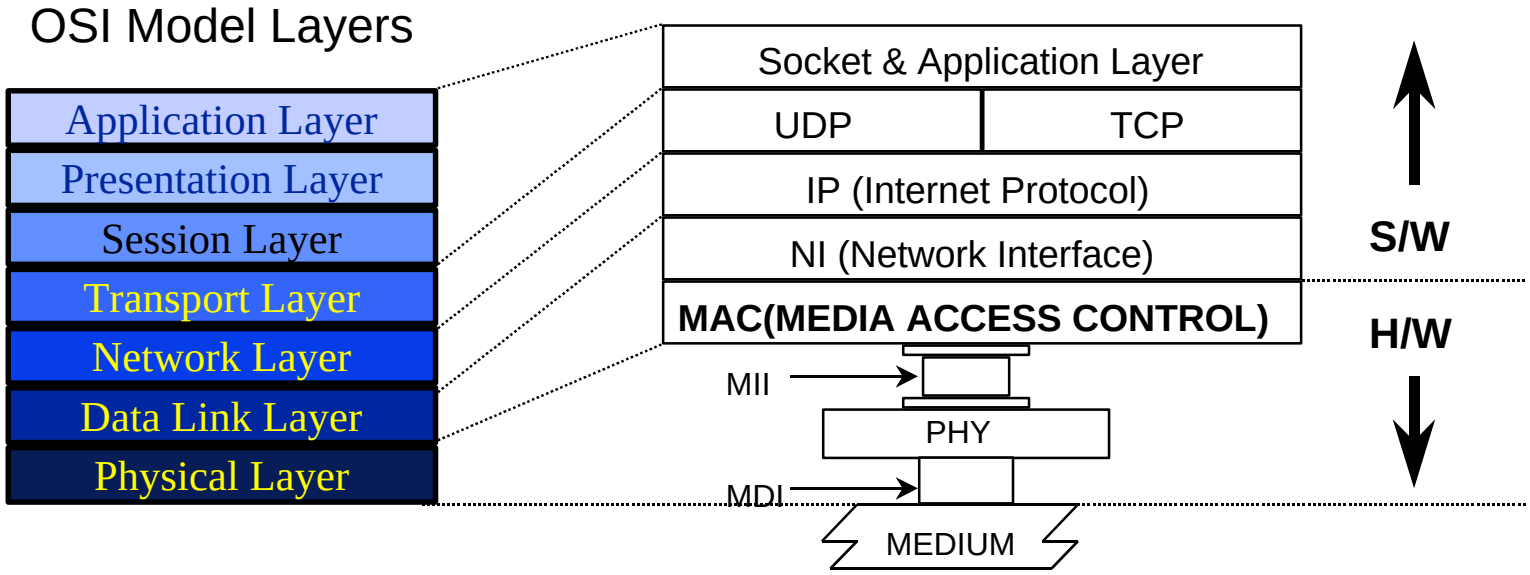
What is Network ?

- **Network Components**
  - **Hosts**
    - Any computing system that is attached to an internet
  - Networks
    - Collection of two or more hosts that are interconnected using a particular form of data link technology
  - **Router**
    - The device that provides connectivity between the various indivisual networks

- **Physical Network Technologies**
  - **Local Area Network (LAN)**
    - High speed, short distance
    - Ethernet, Token Ring, FDDI
  - Wide Area Network (WAN)
    - Low speed, global networks, long distance
    - X.25, ISDN with HDLC

- **OSI Reference Model**
  - **7 Layer for provides connectivity between the various indivisual networks**

- **TCP/IP Networking Software**
  - TCP/IP networking software provides a unified interface that is independent of the various indivisual networks



- **Network Interface Layer**
  - H/W MAC Driver
  - ARP (Address Resolution Protocol)
    - Mapping Internet address to physical network address
  - RARP (Reverse Address Resolution Protocol)
    - Obtain Internet address from physical network address
- **IP (Internet Protocol) Layer**
  - Routing, Fragmentation, Reassembly of Datagrams
  - ICMP Protocol
    - Error report and network management tasks
- **Transport Layer**
  - TCP (Transmission Control Protocol)
    - Deliver packet by connection-oriented method
  - UDP (User Datagram Protocol)
    - Deliver packet by connectionless method
- **Socket Layer**
  - Application Programming Interface
- **Application Layer**
  - TFTP, FTP, TELNET, DNS, NFS, RPC, SMTP, SNMP

- **Why we need RTOS ?**
  - Task management
  - Memory allocation
  - Interrupt completion service
  - Easy to develop application system that has network interface

- **What kind of RTOS is supported for SAMSUNG's NetMCU ?**
  - pSOS+ (ISI)
  - Nucleus (ATI)

- **Where can we get H/W device driver for RTOS ?**
  - Samsung WEB-Site : www.samsungsemi.com

- **How can we use the H/W device driver ?**
  - After download the H/W device driver from Samsung WEB site, you should extract and rebuild again for your purpose

pSOSystem Components

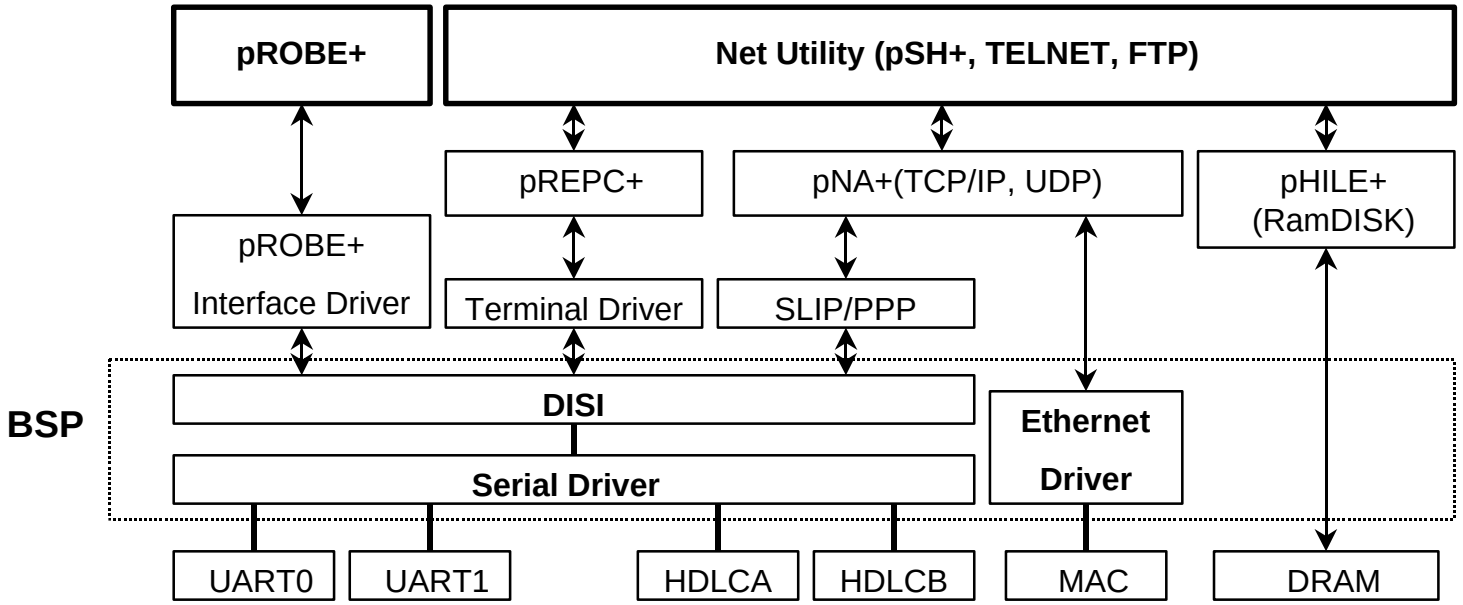
- pSOS+ : Single Processor Kernel
- pSOS+m : Multiprocessor Kernel
- pROBE+ : Target Based Debugger
- pHILE+ : File Management
- pNA+ : TCP/IP Networking
- PPP
- pREPC+ : ANSI C Run-time Library
- Drivers/Board Support Package(BSP)

pSOSystem Debugging Environments

- S/W Debugging Environments
  - pROBE+ ROM : Target Based Debugger interface ROM
    - ➲ Customer can get from **Samsung WEB site and ISI**
  - pRISM+ : Debugger interface running on host system
    - ➲ Customer should pay charge to **ISI**
- H/W Debugging Environments
  - Embedded-ICE
    - ➲ Customor can get from **ARM agent**
  - ARM SDT(Software Development Toolkit)
    - ➲ Customor can get from **ARM agent**

What is pSOSystem BSP(Board Support Package) ?

- **H/W device driver for pSOS system**
  - Timer
  - Serial Driver (UART, HDLC)
  - MAC Driver



Nucleus Components

- Kernel
- NET4.0 : TCP/IP Protocol stack
- Extended Protocol Package for Nucleus NET
- PPP
- File System

Nucleus Debugging Environments

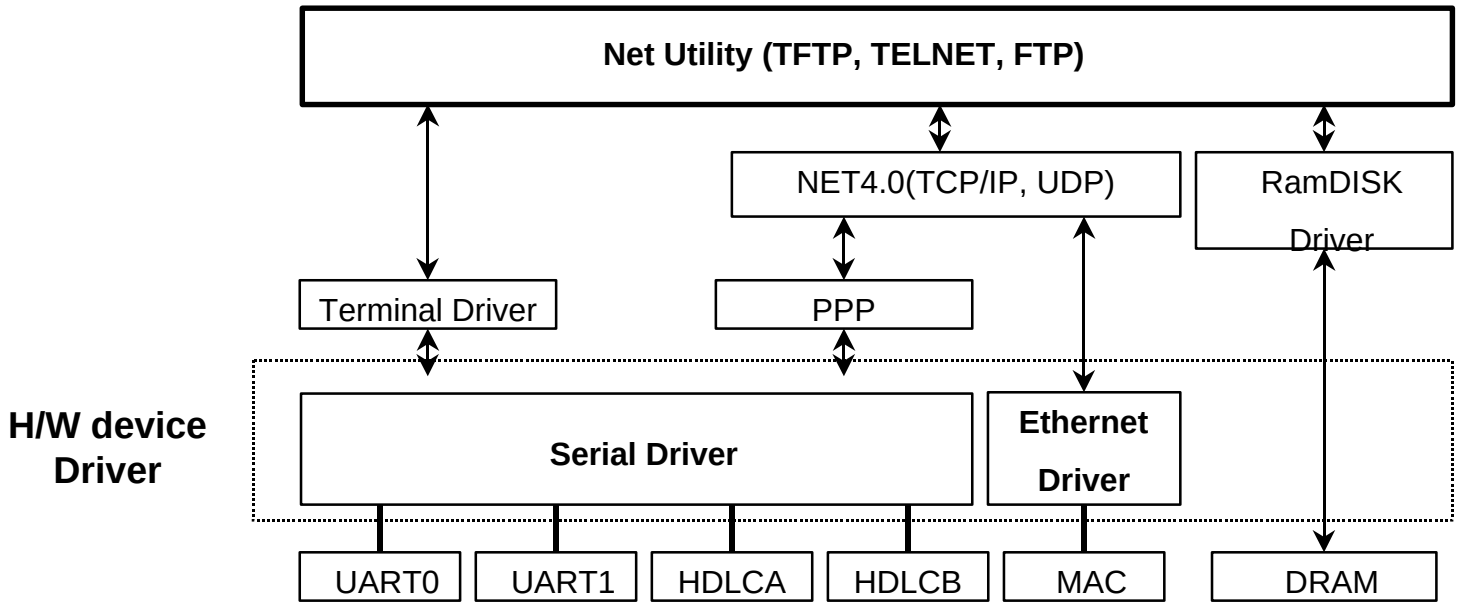
- S/W Debugging Environments
  - UDB
    - ➲ Not supported yet
- H/W Debugging Environments
  - Embedded-ICE
    - ➲ Customor can get from **ARM agent**
  - ARM SDT(Software Development Toolkit)
    - ➲ Customor can get from **ARM agent**

Supports

- Samsung : H/W Device Driver for Nucleus
- ATI : All Nucleus stack
- Application : Application designer (Customer side)

H/W device Driver for Nucleus

- Timer
- Serial Driver (UART, HDLC)
- MAC Driver

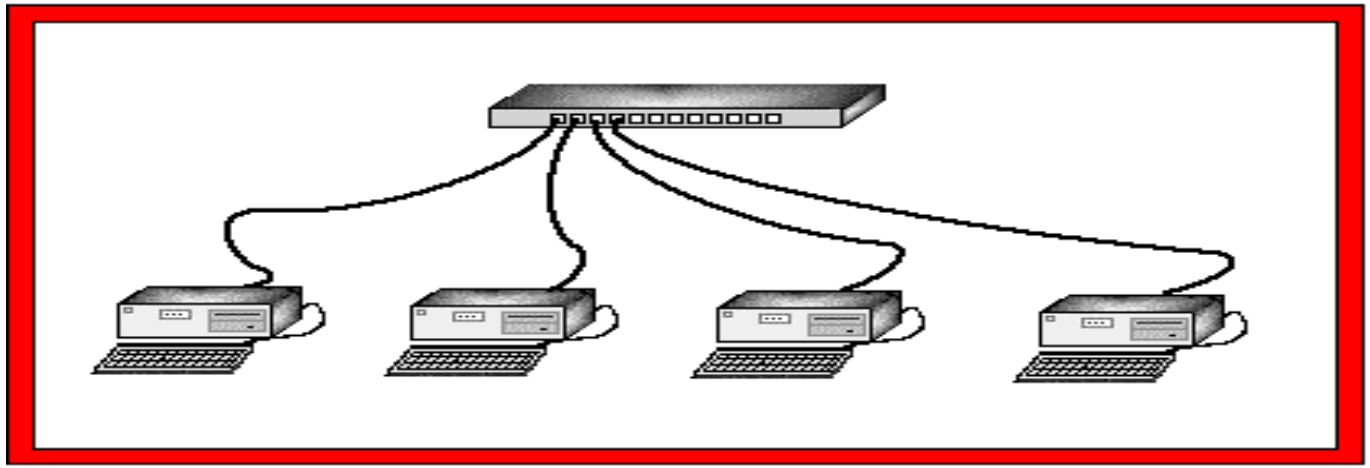


- PMAKE

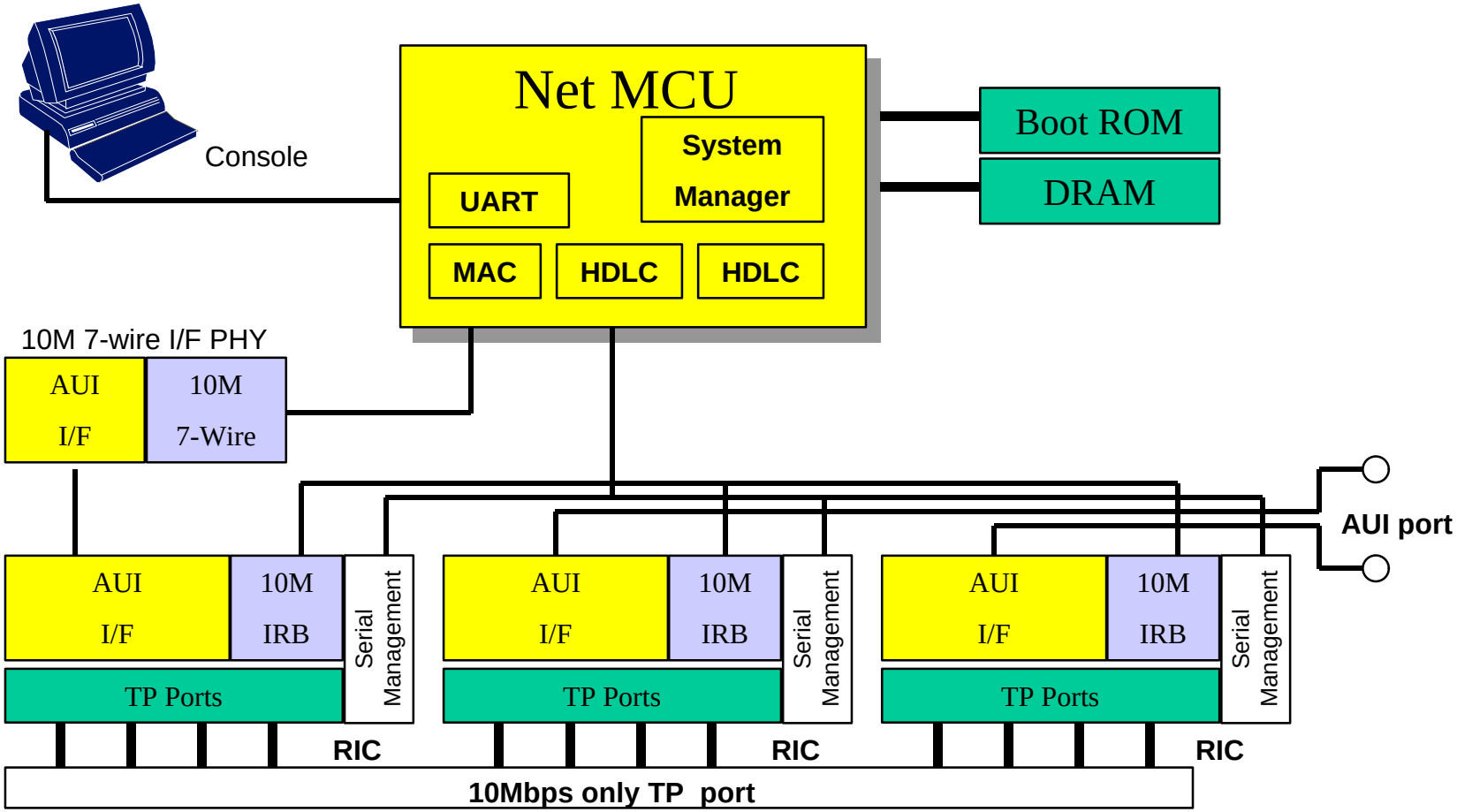
Applicable System with SAMSUNG's NetMCU

- **Managed HUB**
- **Managed Switching HUB**
- **Router / Layer-3 Switching**
  - Modem Router
  - IP Router / IP Sharing
  - ISDN Router
  - ADSL Router
- **Printer Server**
- **Network Printer**
- **Cable Modem**
- **UPS Management Controller**

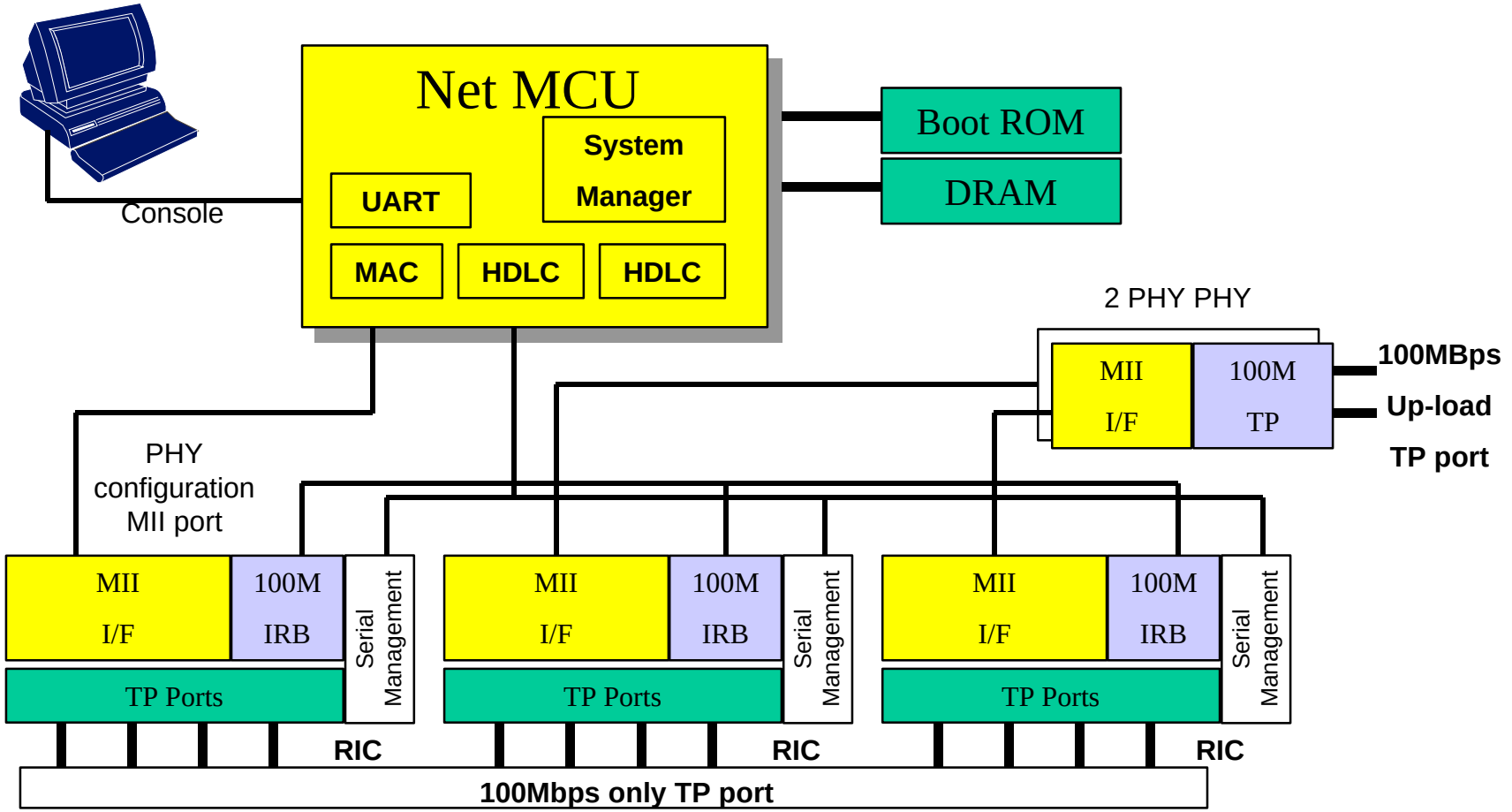
- **Network Topology**
  - The physical and/or electrical configuration of cabling and connections comprising a network -- the shape of the system.
  - Bus, Star, Mesh, Ring, Star
- **Star Topology**
  - Most popular
  - each device has its own cable run connecting the device to a common hub or concentrator. Only one device is permitted to use each port on the hub.



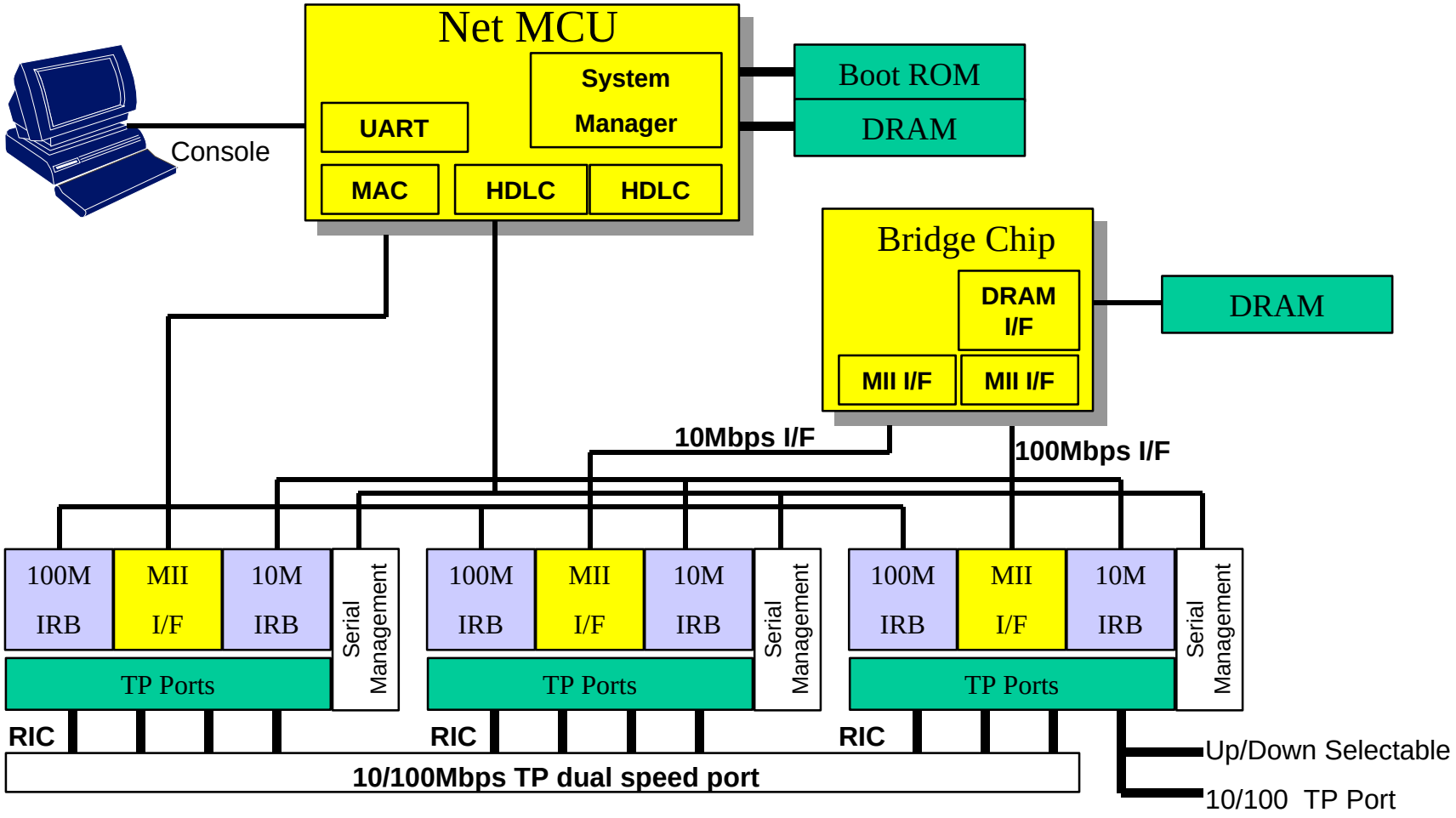
Excellence in Low-Power The way MICOM/DSP should be



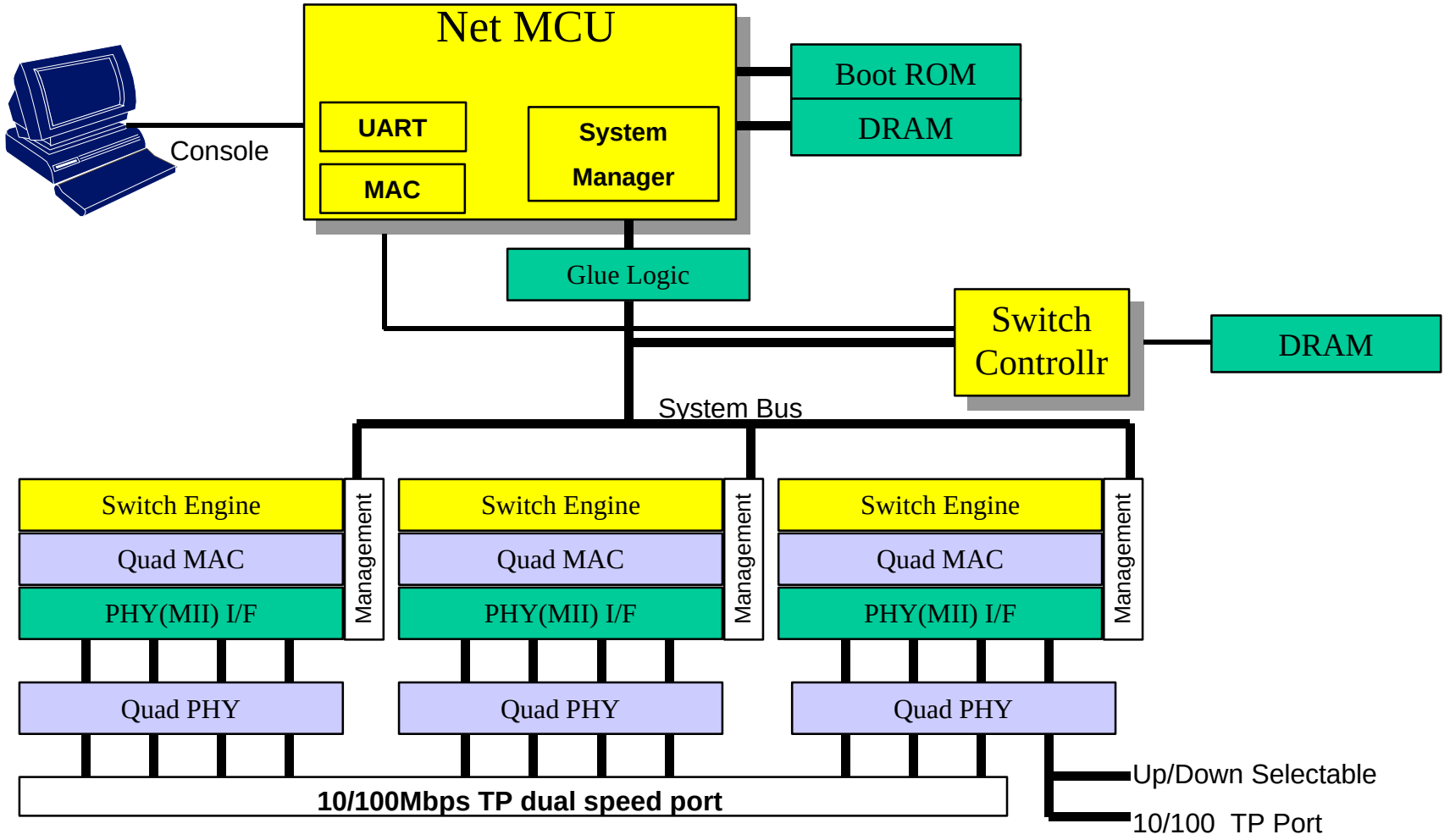
Excellence in Low-Power *The way MICOM/DSP should be*



Excellence in Low-Power The way MICOM/DSP should be



Excellence in Low-Power The way MICOM/DSP should be



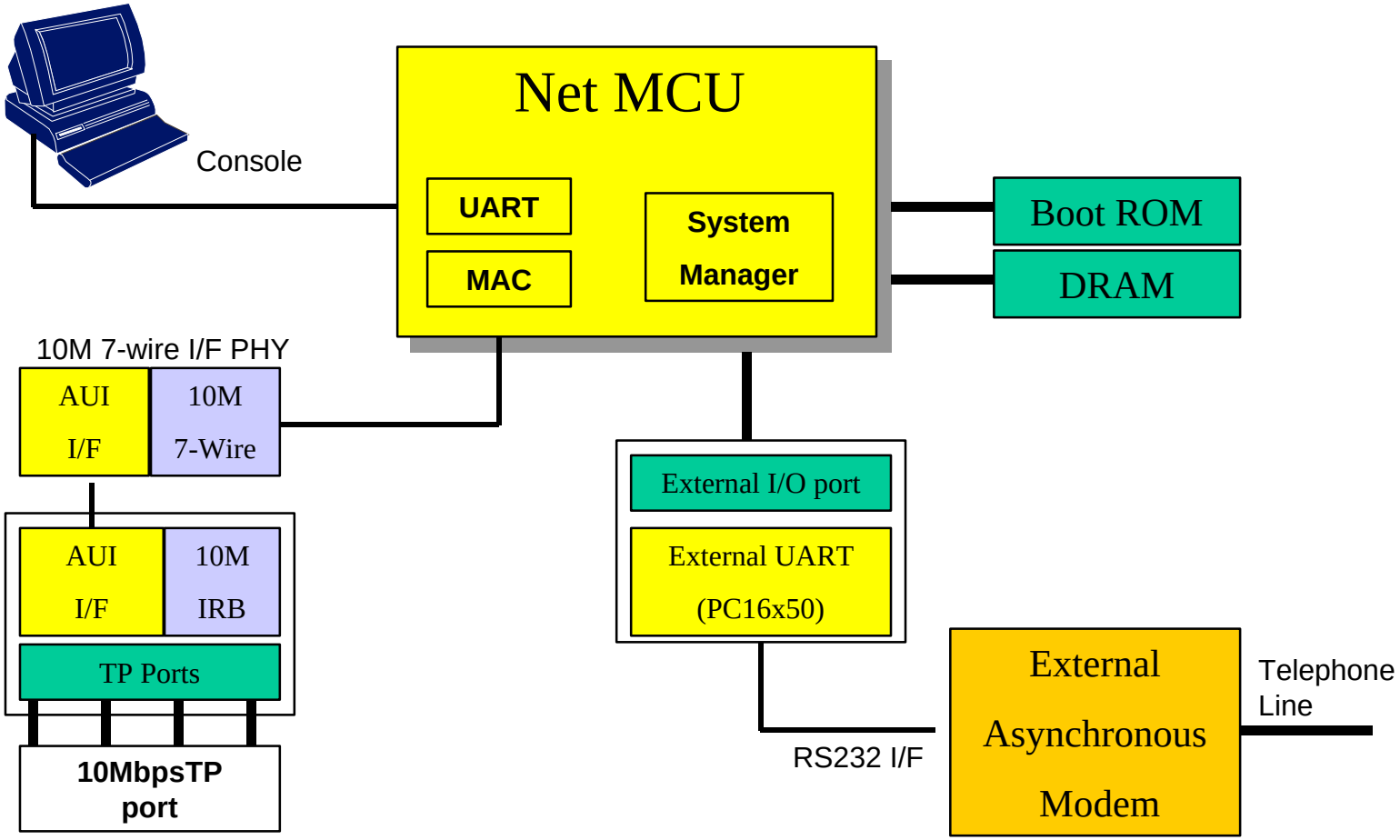
Function

- Multiple LAN users to access the Internet simultaneously, using a single IP address through a 33.6k/56kb or ISDN modems.
- World Wide Web (WWW) for setup with your Router
- Supports BOOTP/DHCP for automatic IP address assignment
- Standard 10/100 BaseT network interface with 4 port HUB.

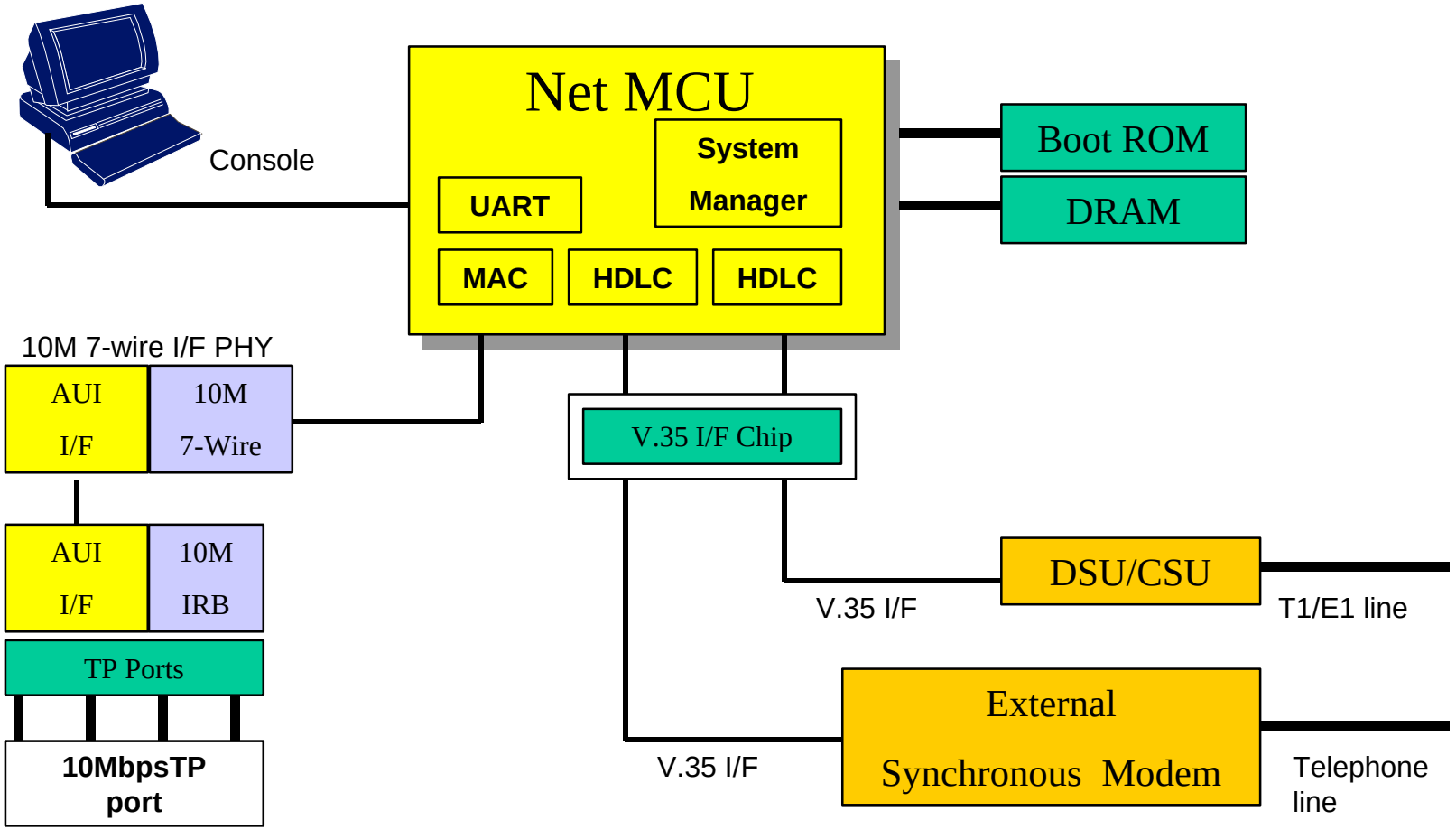
Software for Router

- TCP/IP stack
  - UDP
  - TCP
  - ICMP
  - ARP/RARP
- Routing Database
- RIP
- DNS resolver
- Packet filtering
- Network Address Translation (NAT)
- PPP (PAP/CHAP/LCP), ML-PPP
- Dynamic/static IP support
- SNMP
- HTTP
- BOOTP/DHCP

Excellence in Low-Power The way MICOM/DSP should be

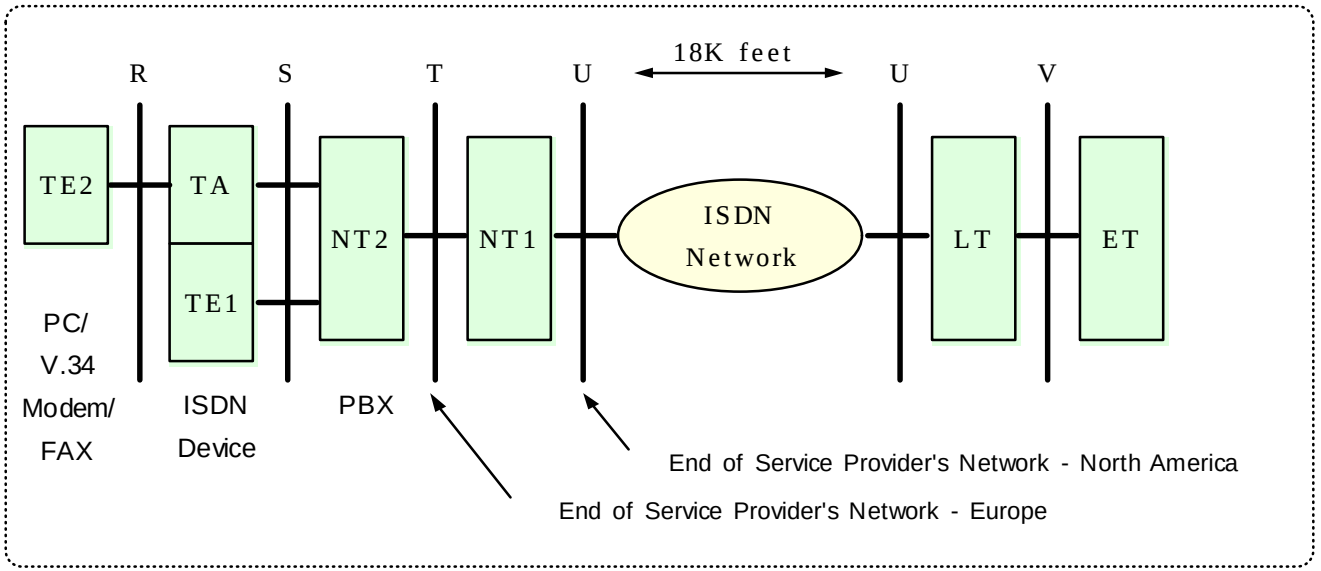


Excellence in Low-Power The way MICOM/DSP should be



ISDN Reference Model

Excellence in Low-Power The way MICOM/DSP should be



ISDN Reference Point

V : Proprietary interface within Central Office

U : 2-wire Interface up to 18K feets

S/T : 4-wire interface up to 1K meters

R : Any non-ISDN interface (RS-232, V.34)

ISDN Reference Equipment

LT(Line Termination) : C.O Switch or Remote line card

NT1/NT2(Network Termination) : CPE connection to network

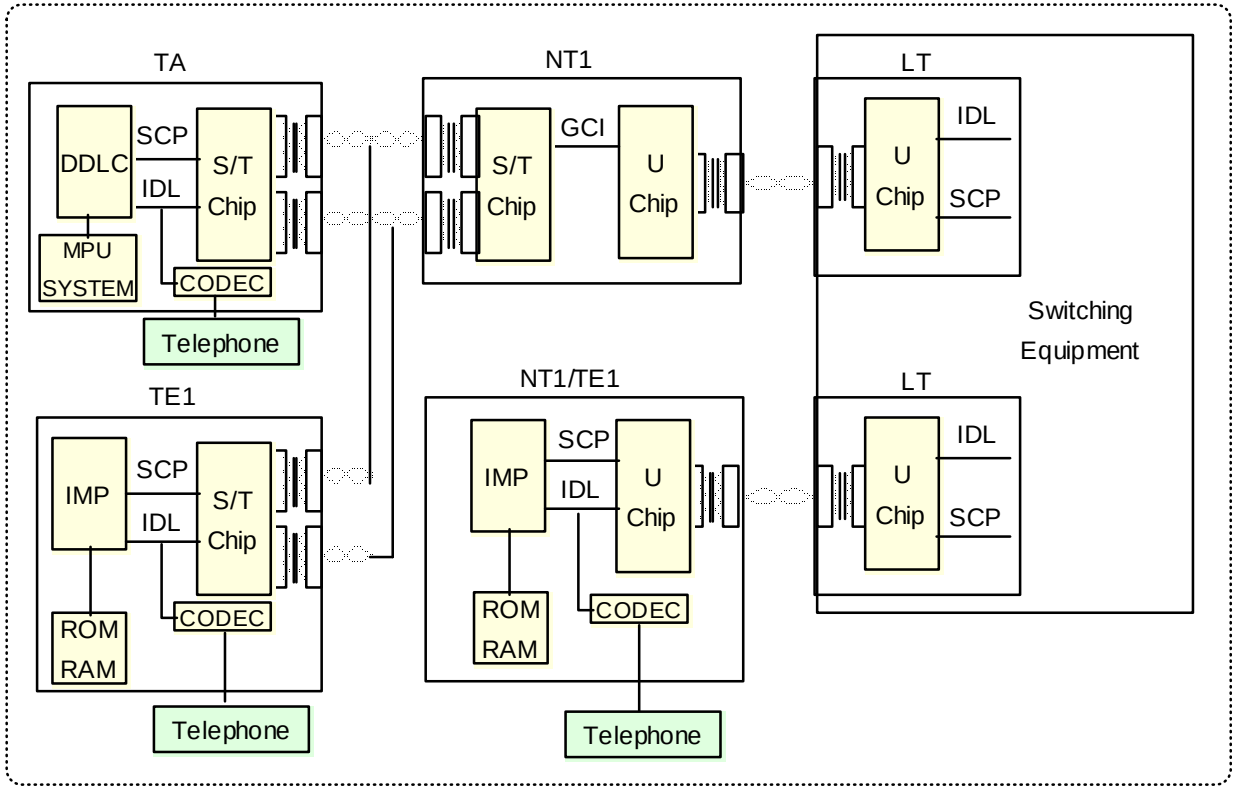
TE1(Terminal Equipment type 1) : ISDN compatable terminal

TE2(Terminal Equipment type 2) : Non-ISDN terminal

TA (Terminal Adapter) : Interface for non-ISDN terminal

ISDN Reference Design (BRI)

Excellence in Low-Power The way MICOM/DSP should be

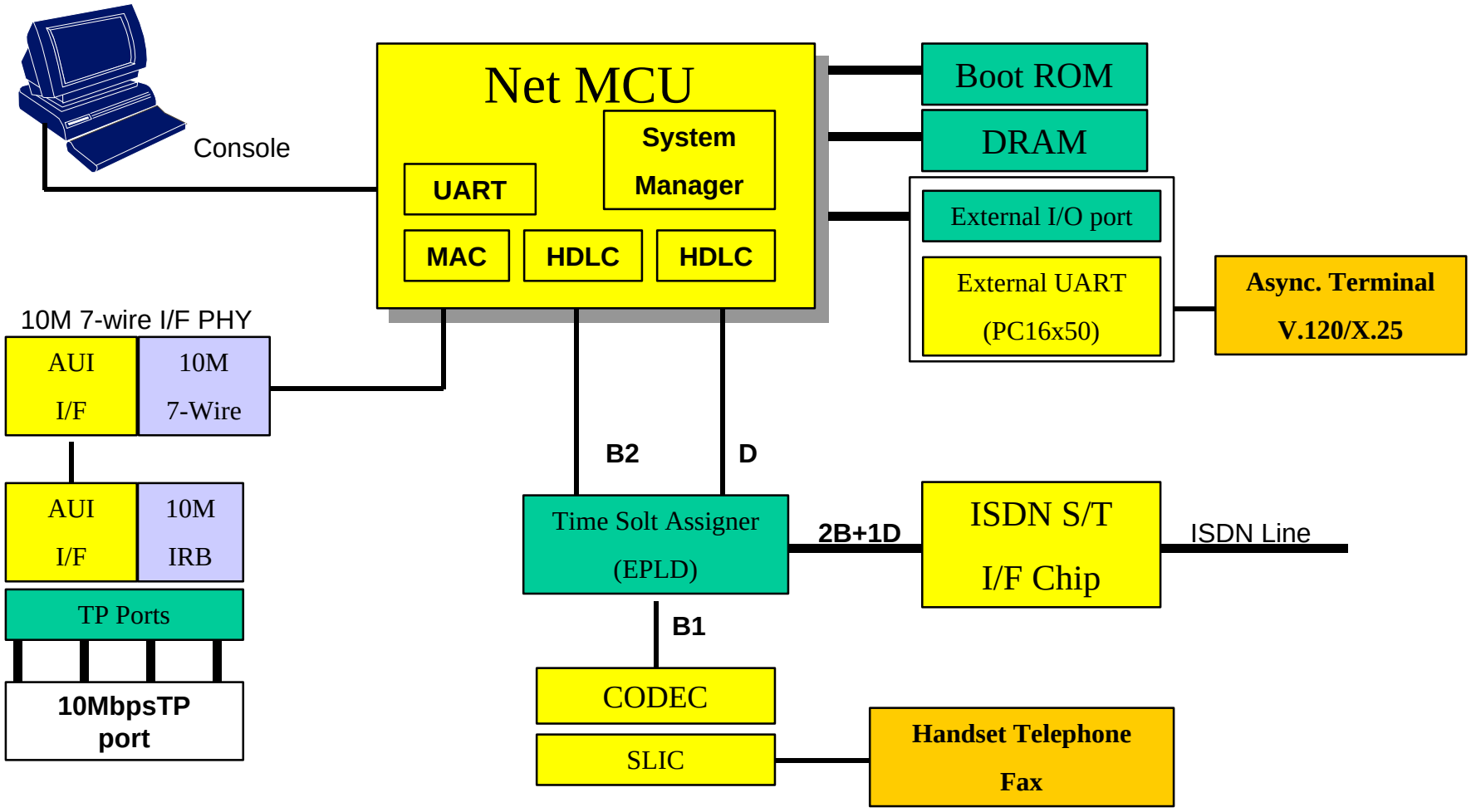


IMP : Integrated Multiprotocol Processor

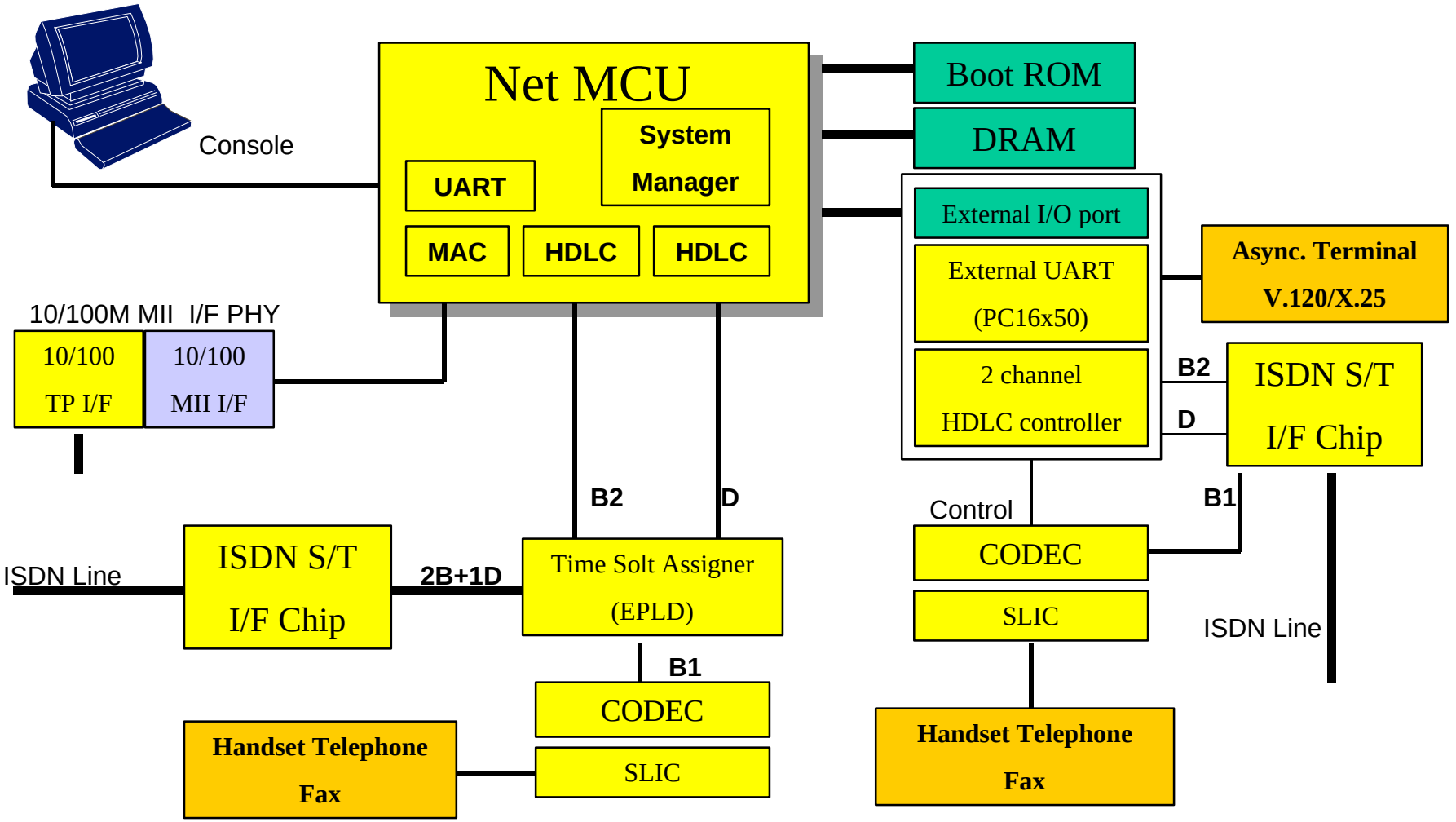
IDL : Motorola Interchip Digital Link

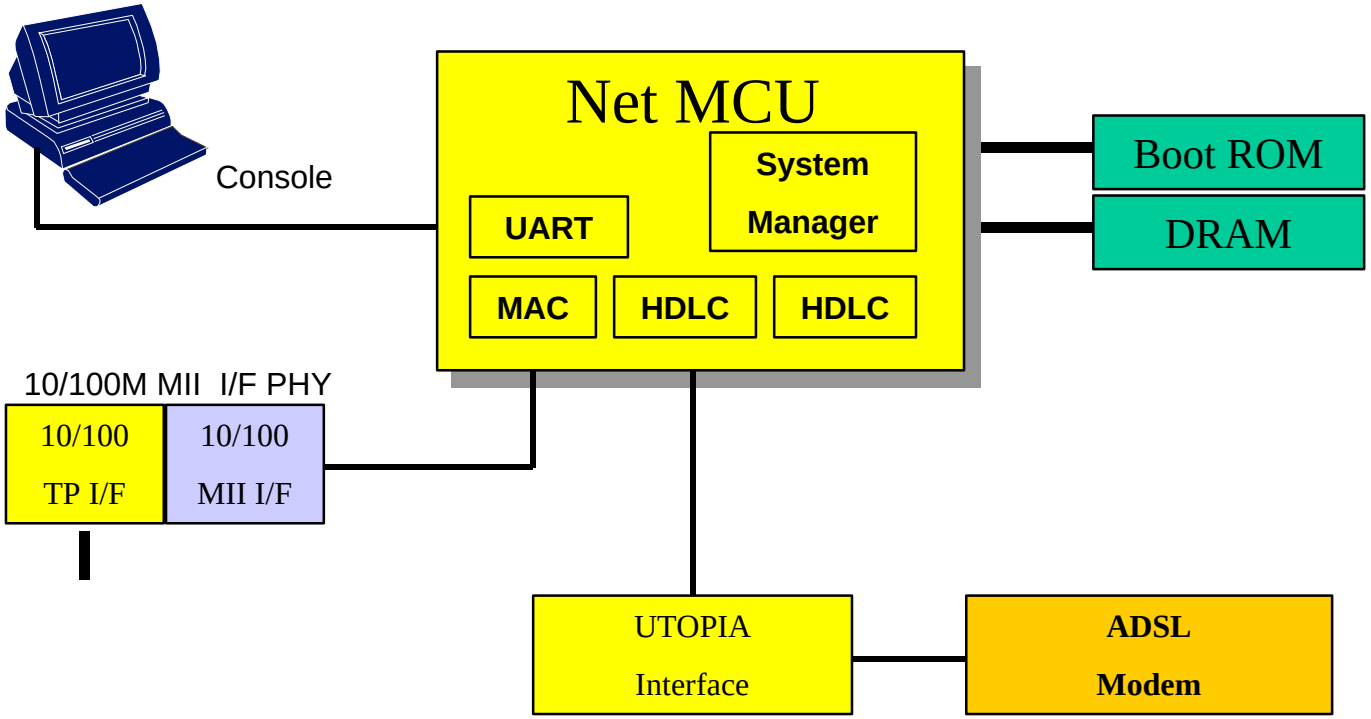
SCP : Serial Communication Port

Excellence in Low-Power The way MICOM/DSP should be

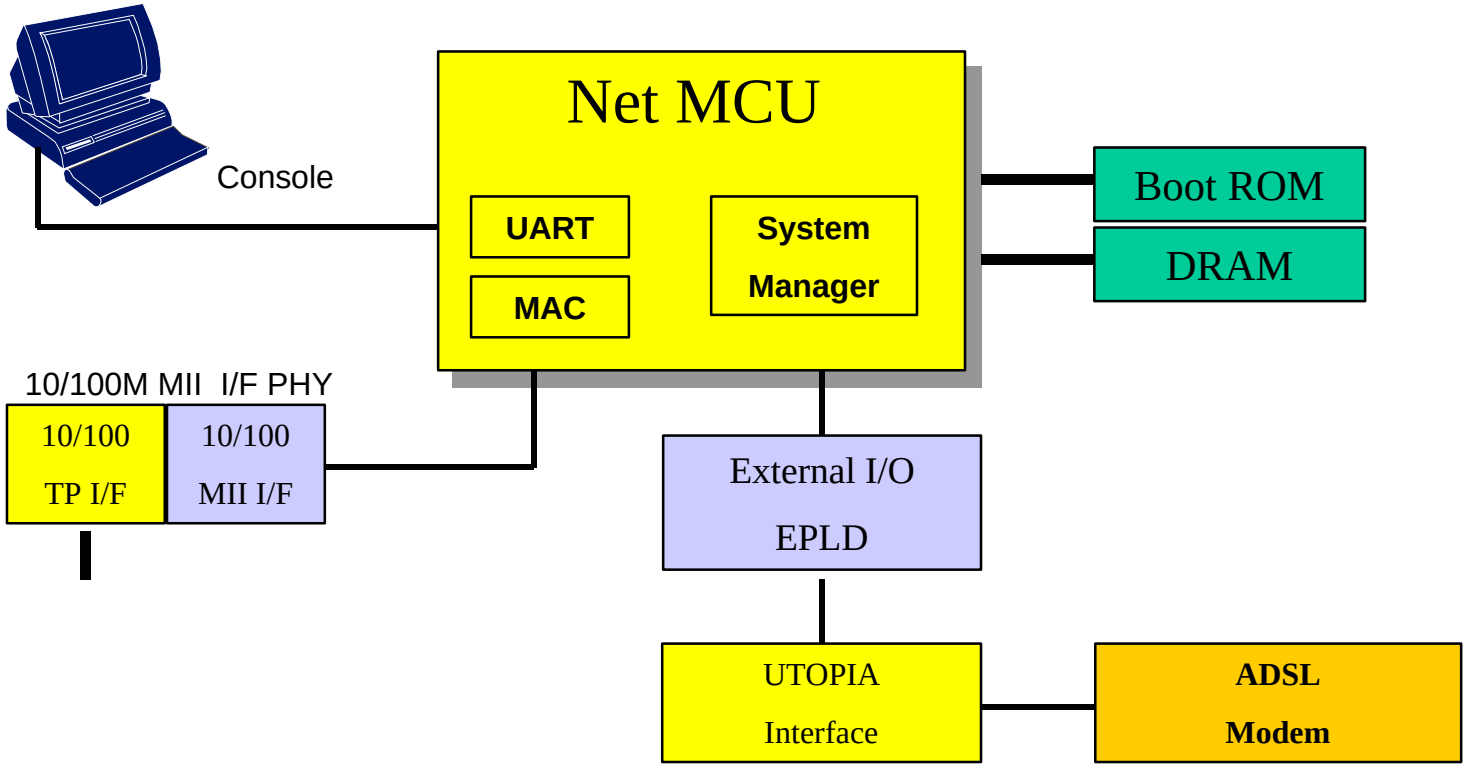


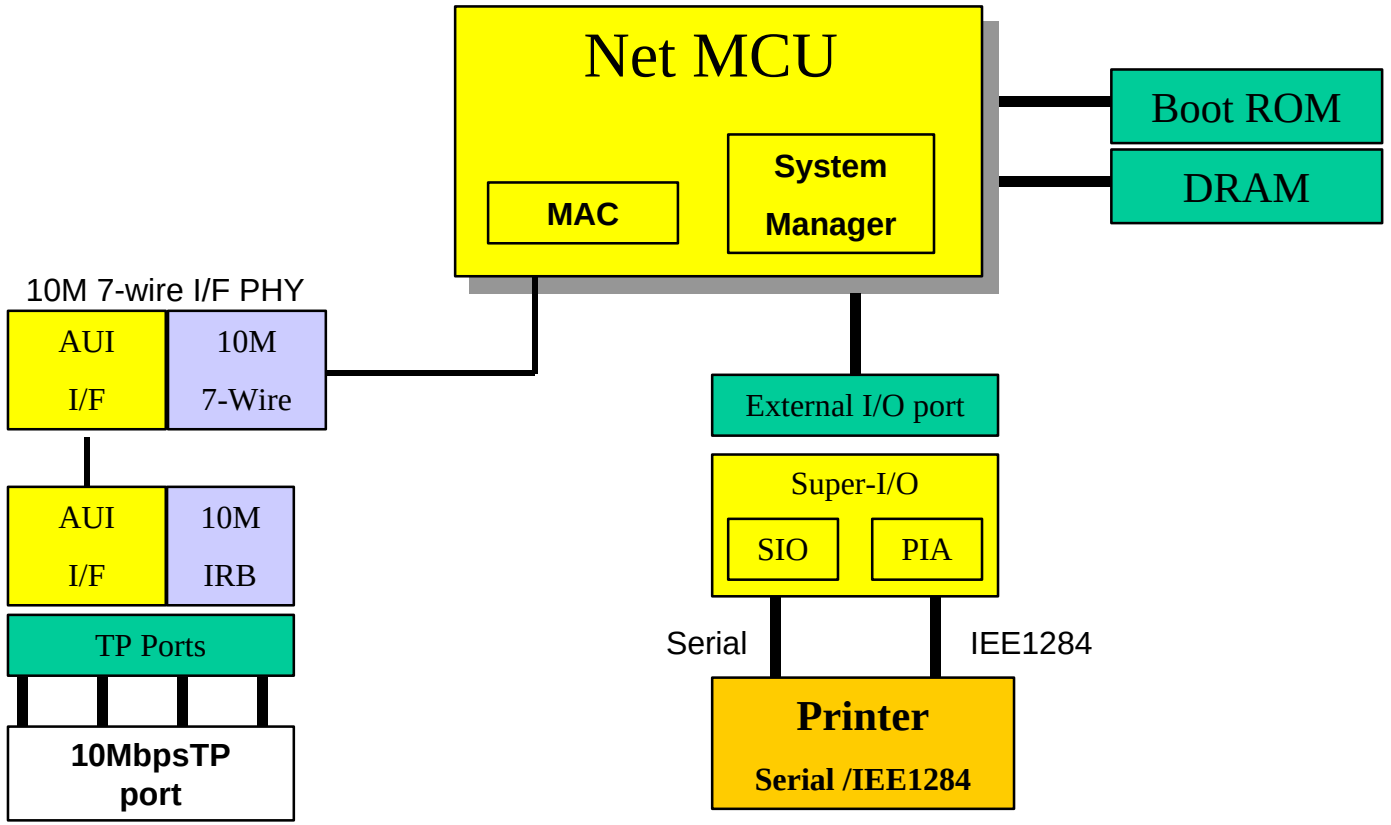
Excellence in Low-Power The way MICOM/DSP should be



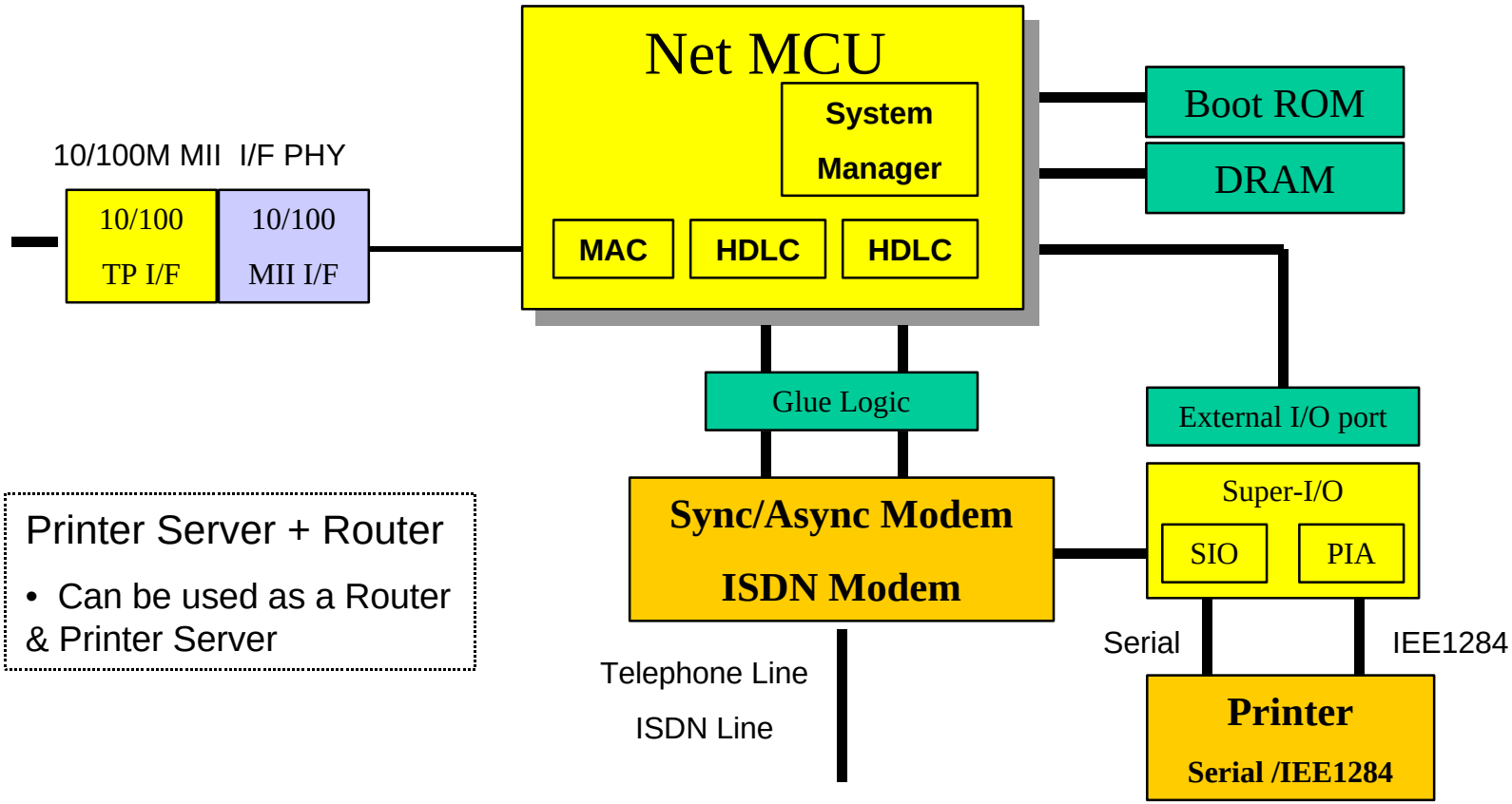

Net MCU
System Manager
UART
MAC
HDLC
HDLC
Console
Boot ROM
DRAM
10/100M MII I/F PHY
10/100 TP I/F
10/100 MII I/F
UTOPIA Interface
ADSL Modem

Excellence in Low-Power The way MICOM/DSP should be




Net MCU
MAC
System Manager
Boot ROM
DRAM
10M 7-wire I/F PHY
AUI I/F
10M 7-Wire
AUI I/F
10M IRB
TP Ports
10MbpsTP port
External I/O port
Super-I/O
SIO
PIA
Serial
IEE1284
Printer
Serial /IEE1284

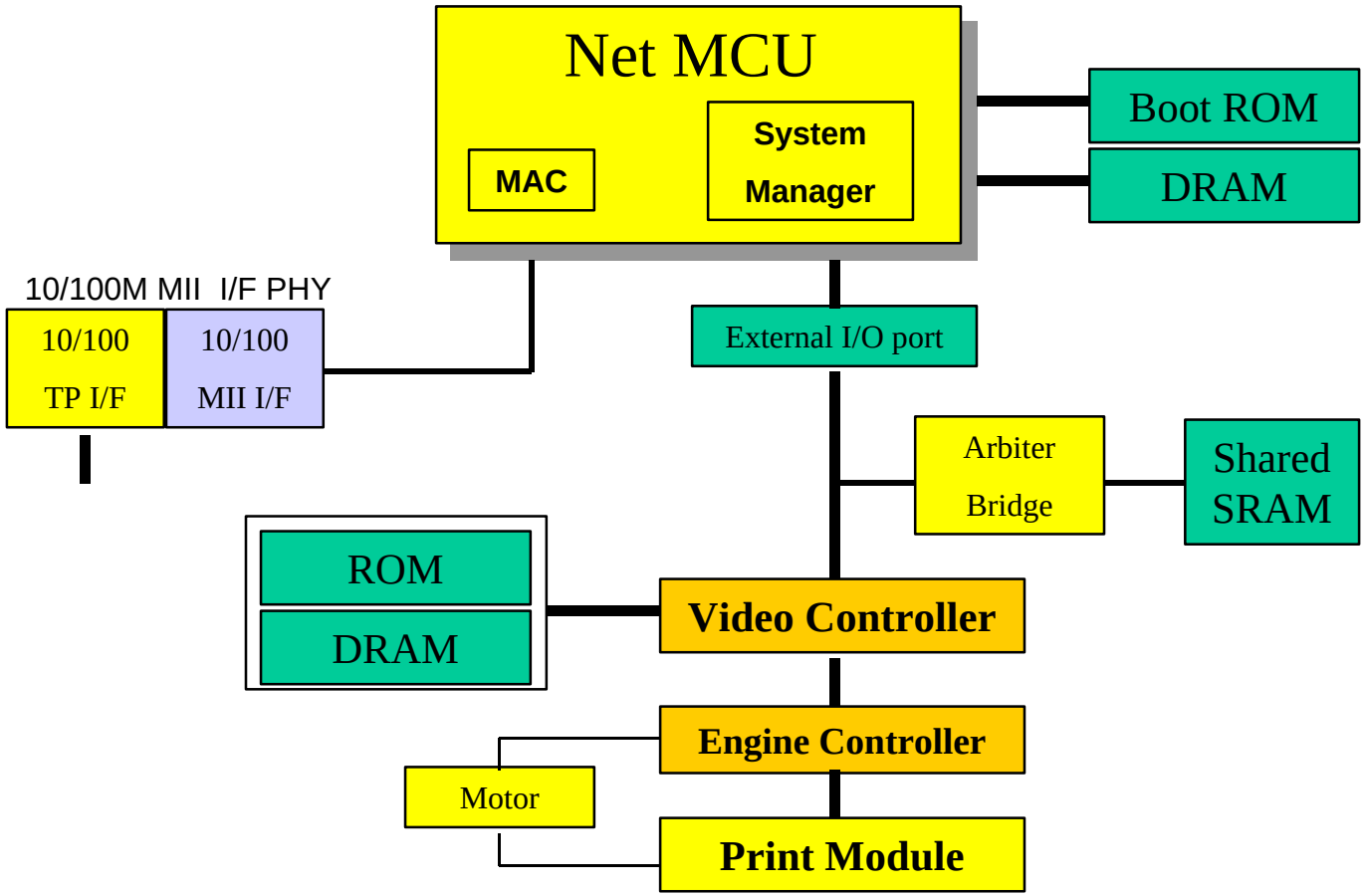
Excellence in Low-Power The way MICOM/DSP should be



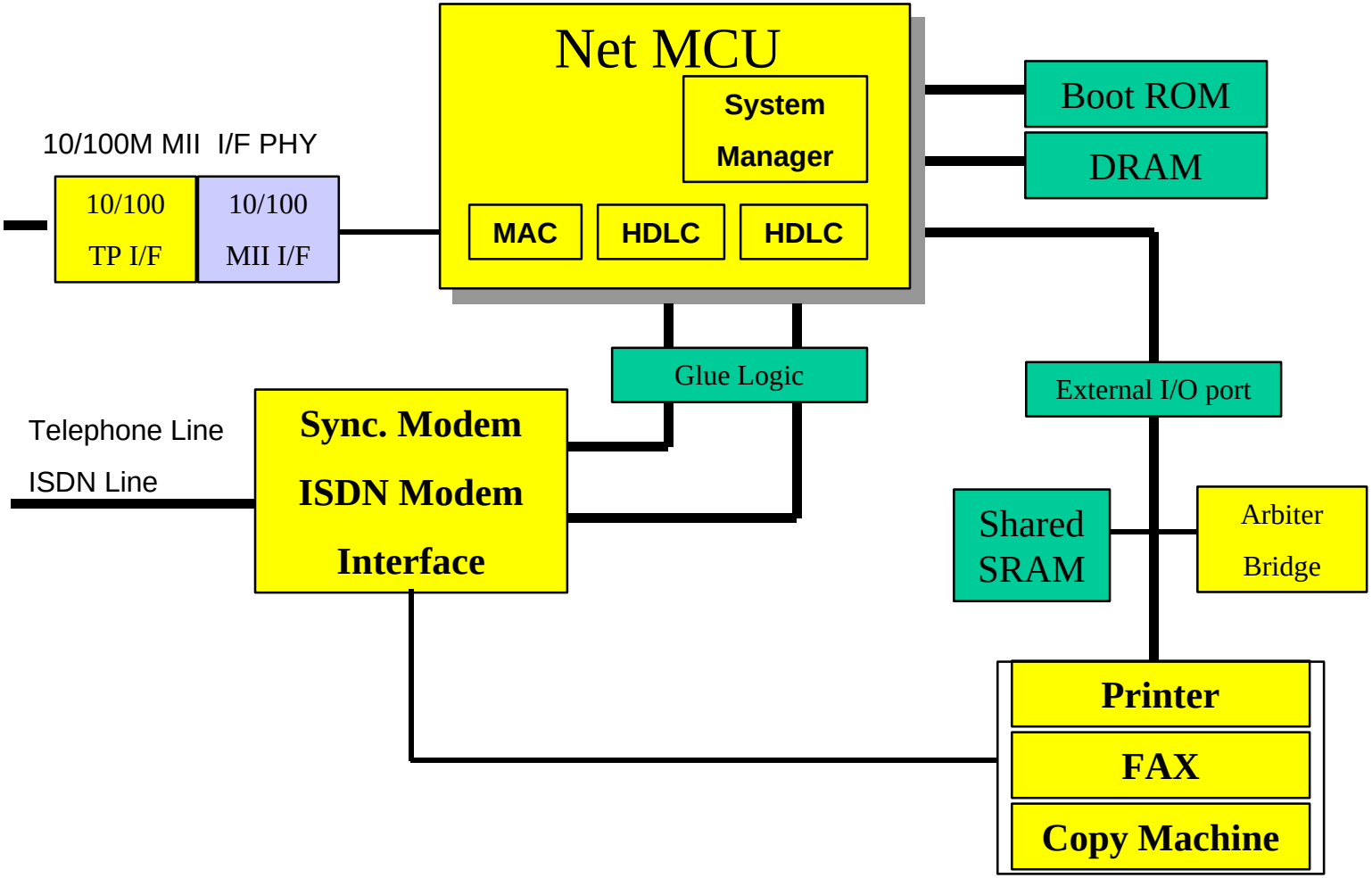
Printer Server + Router

- Can be used as a Router & Printer Server

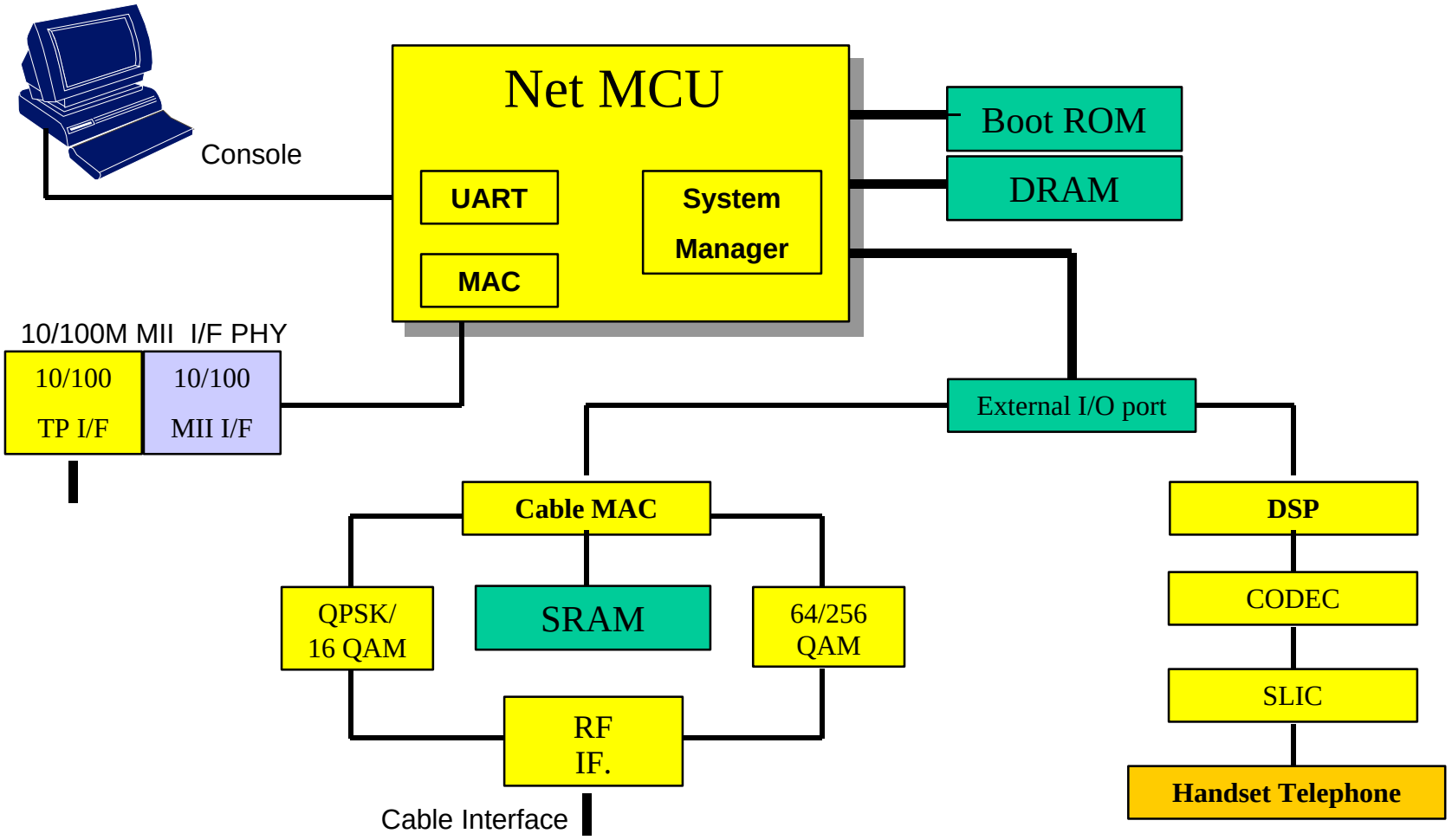
Excellence in Low-Power The way MICOM/DSP should be



Excellence in Low-Power The way MICOM/DSP should be

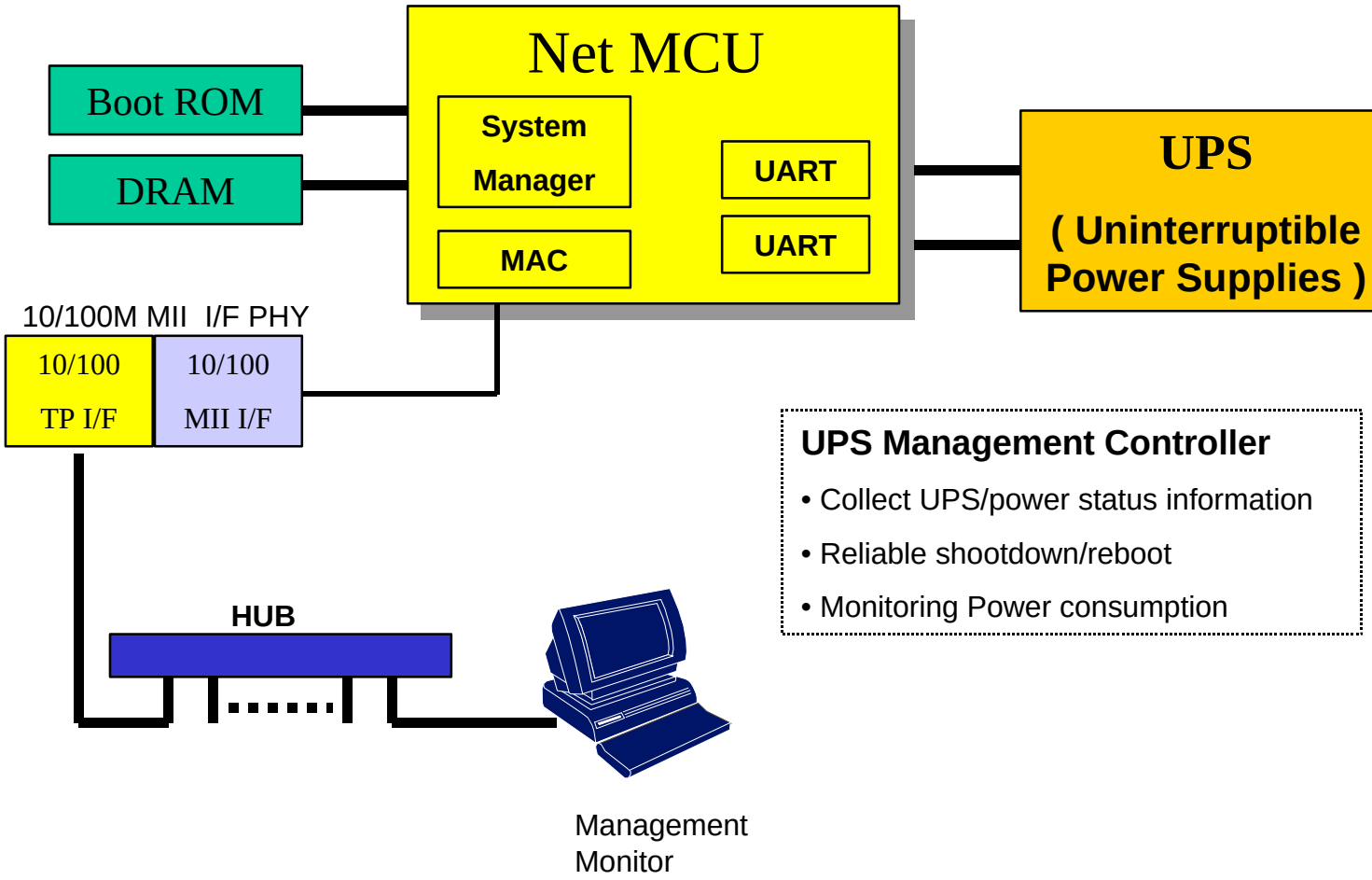


Excellence in Low-Power The way MICOM/DSP should be



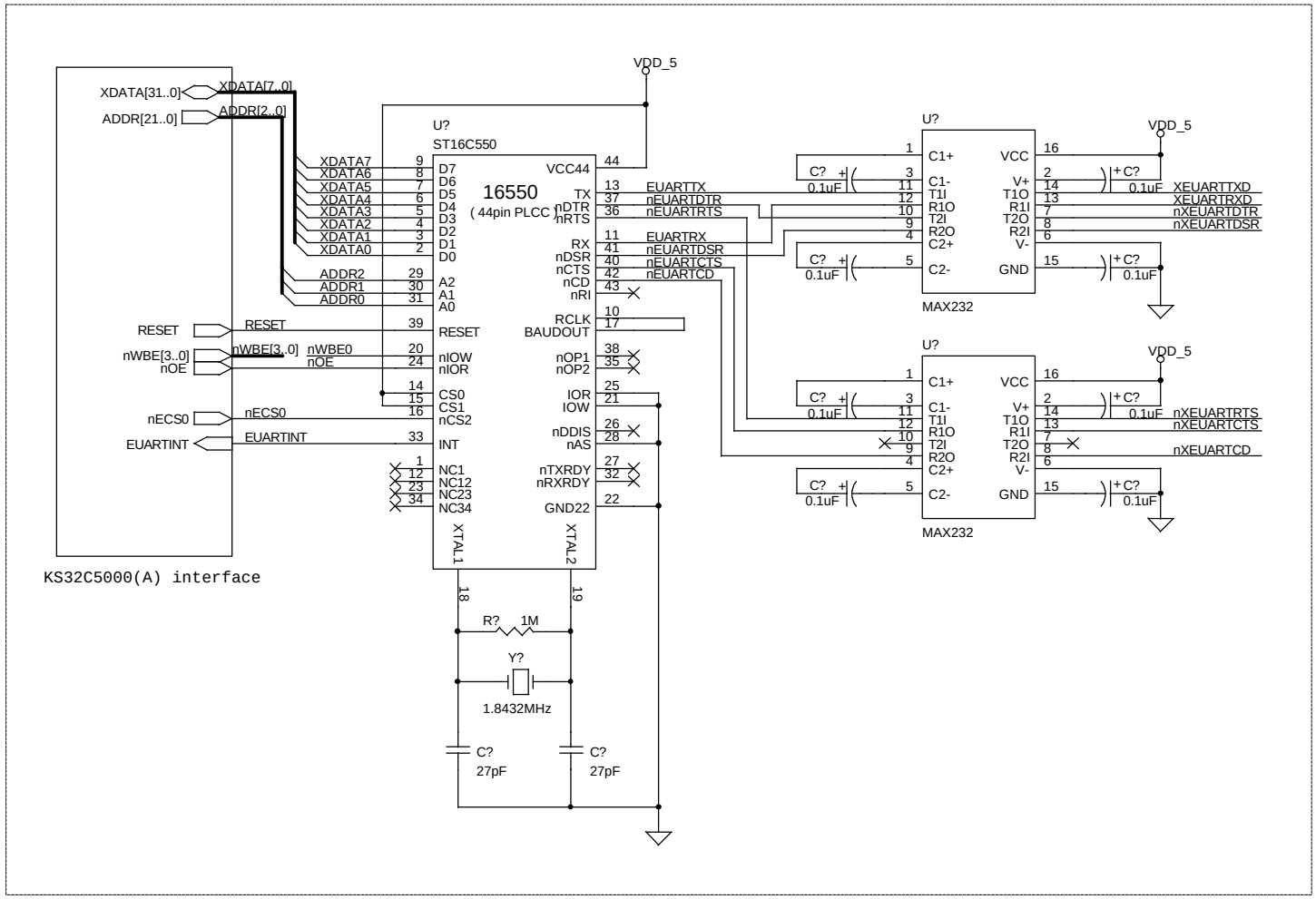
UPS Management Controller

Excellence in Low-Power The way MICOM/DSP should be

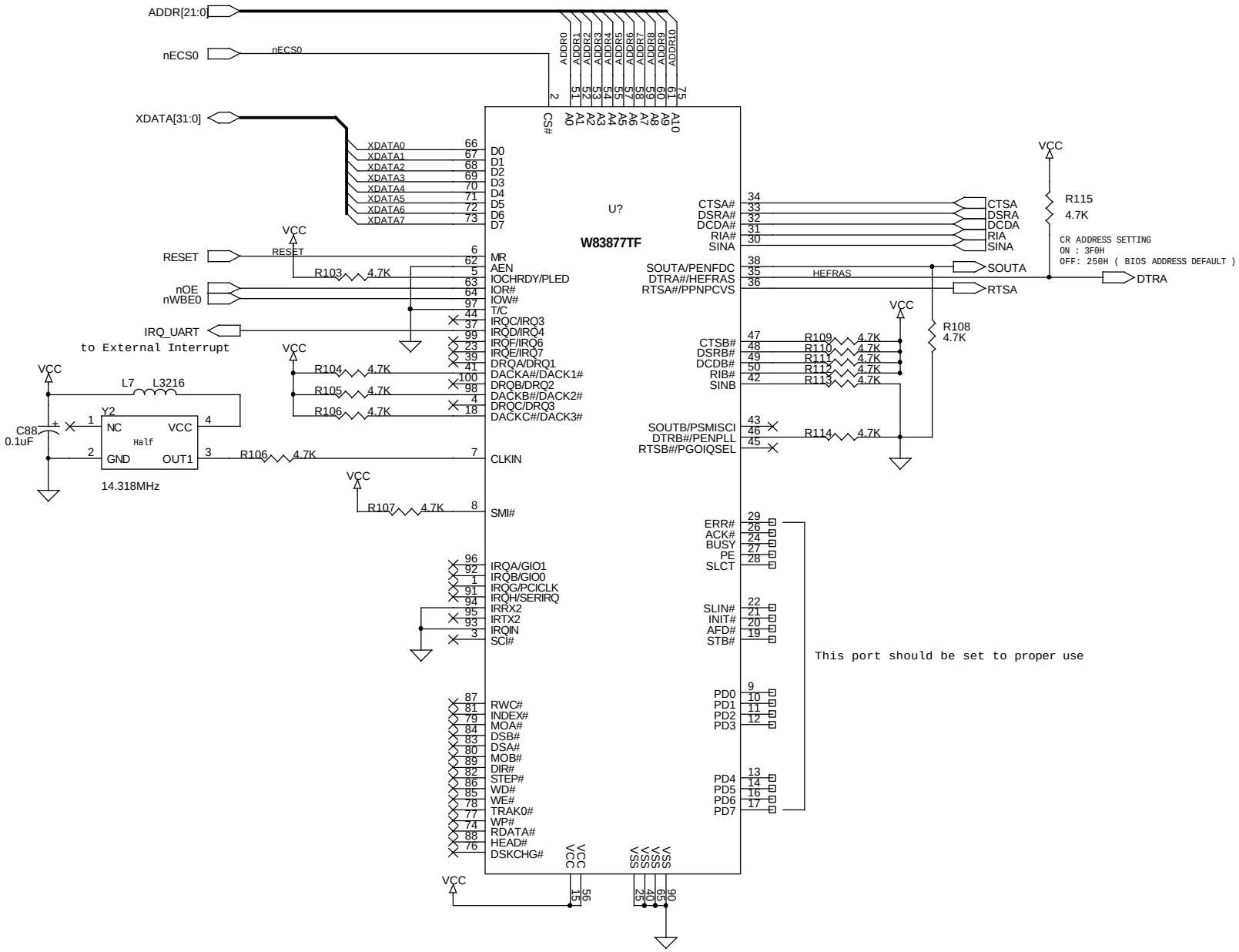


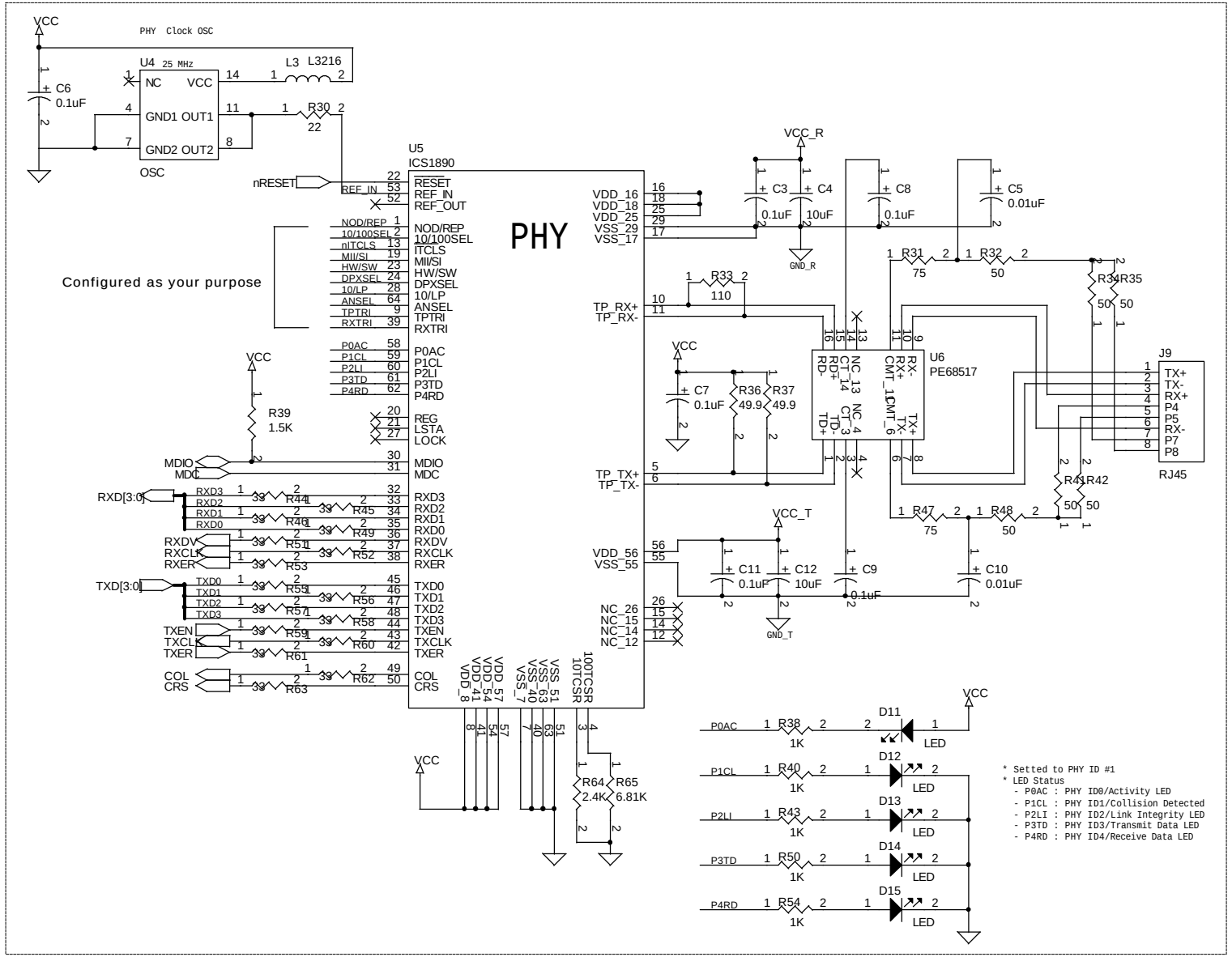
UPS Management Controller

- Collect UPS/power status information
- Reliable shootdown/reboot
- Monitoring Power consumption

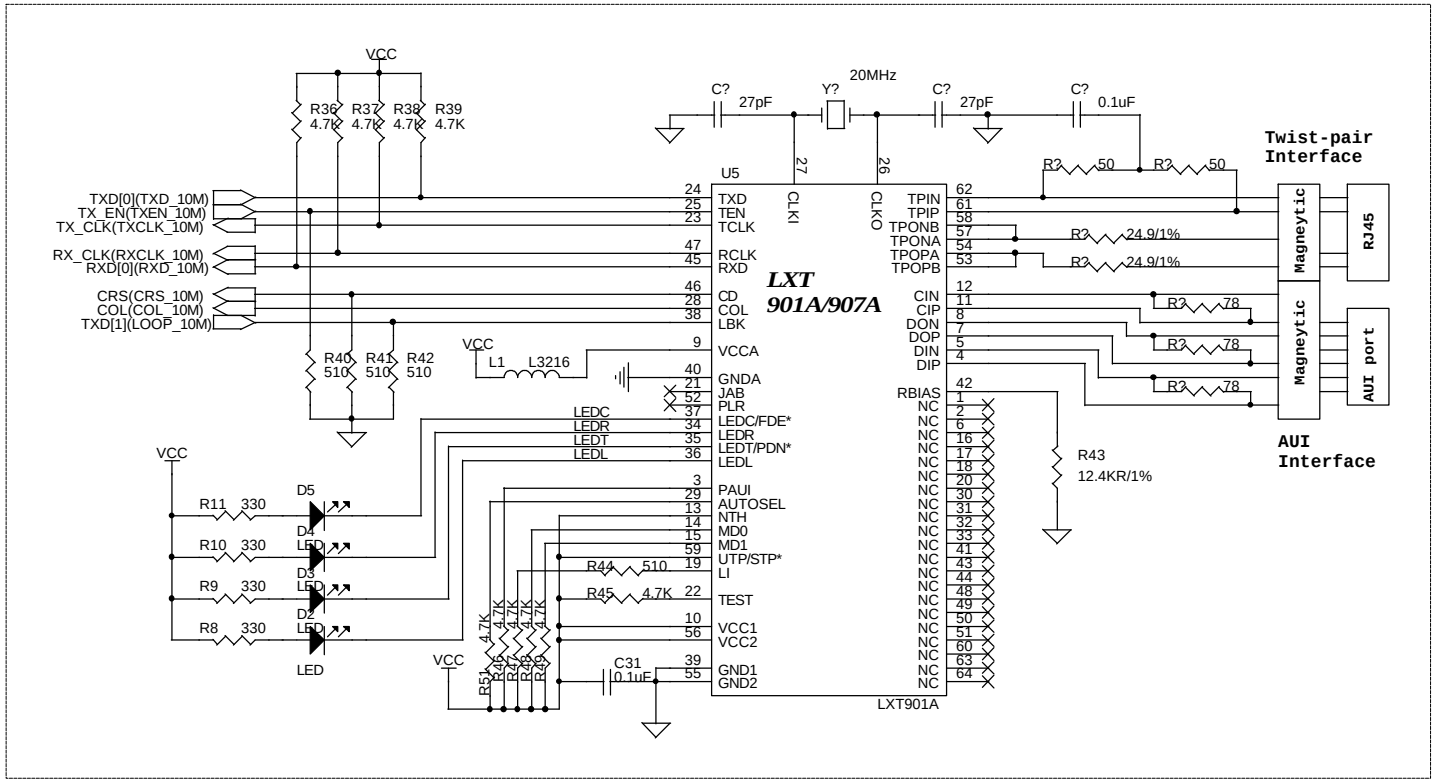


Title	ARM7S BOARD	
Size: B	Document Number: 1	Rev: 0.0
Date: Friday, April 09, 1999	Sheet 1 of 1	

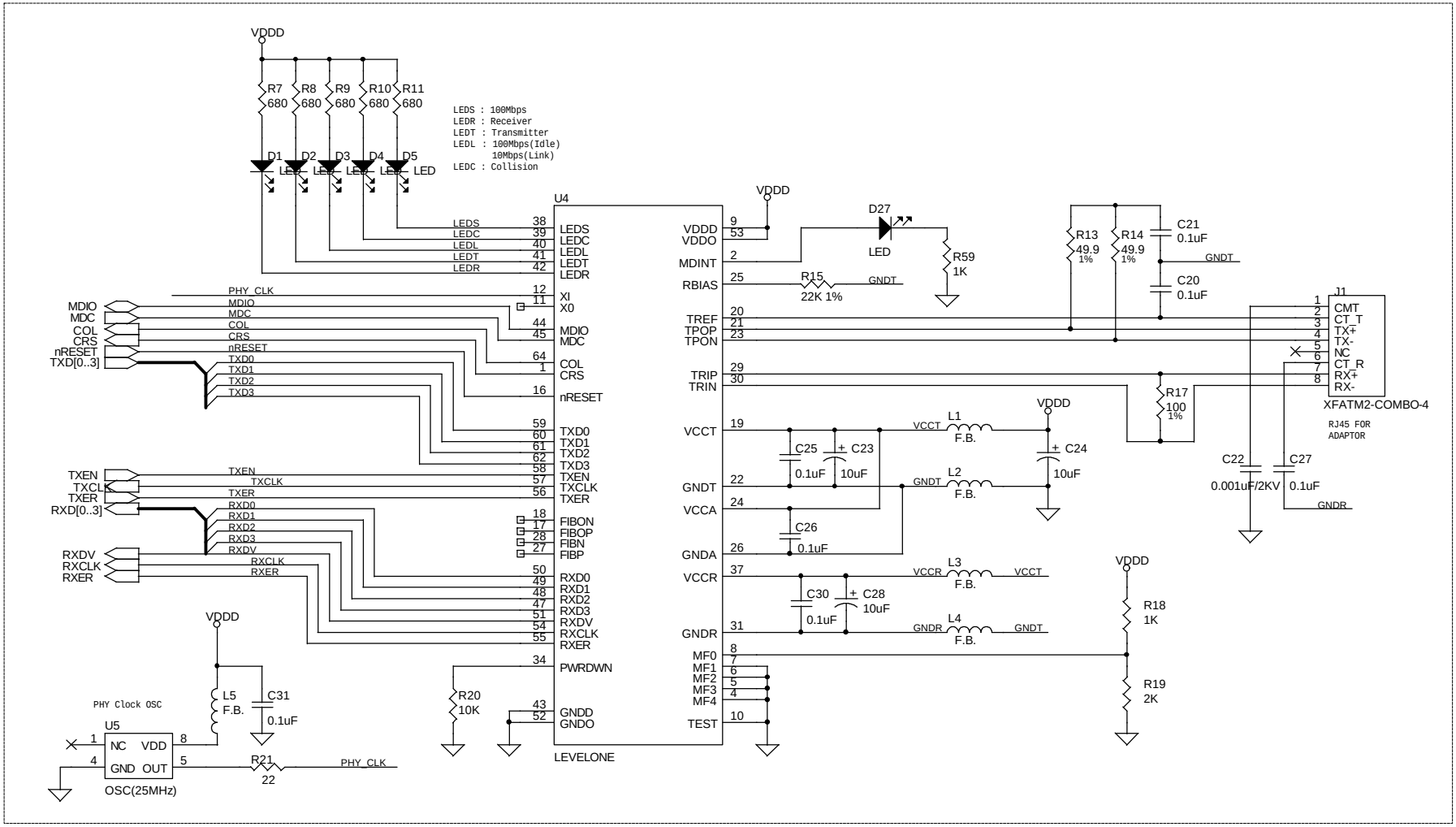

ADDR[21:0]
nECS0
XDATA[31:0]
U?
W83877TF
RESET
nOE
nWBE0
IRQ_UART
to External Interrupt
VCC
L7 L3216
Y2
C88
0.1uF
NC
VCC
GND
OUT1
Half
14.318MHz
R103 4.7K
R104 4.7K
R105 4.7K
R106 4.7K
R107 4.7K
R108 4.7K
R109 4.7K
R110 4.7K
R111 4.7K
R112 4.7K
R113 4.7K
R114 4.7K
R115 4.7K
CR ADDRESS SETTING
ON : 3F0H
OFF: 250H (BIOS ADDRESS DEFAULT)
HEFRAS
CTSA
DSRA
DCDA
RIA
SINA
SOUTA
DTRA
RTSA
This port should be set to proper use



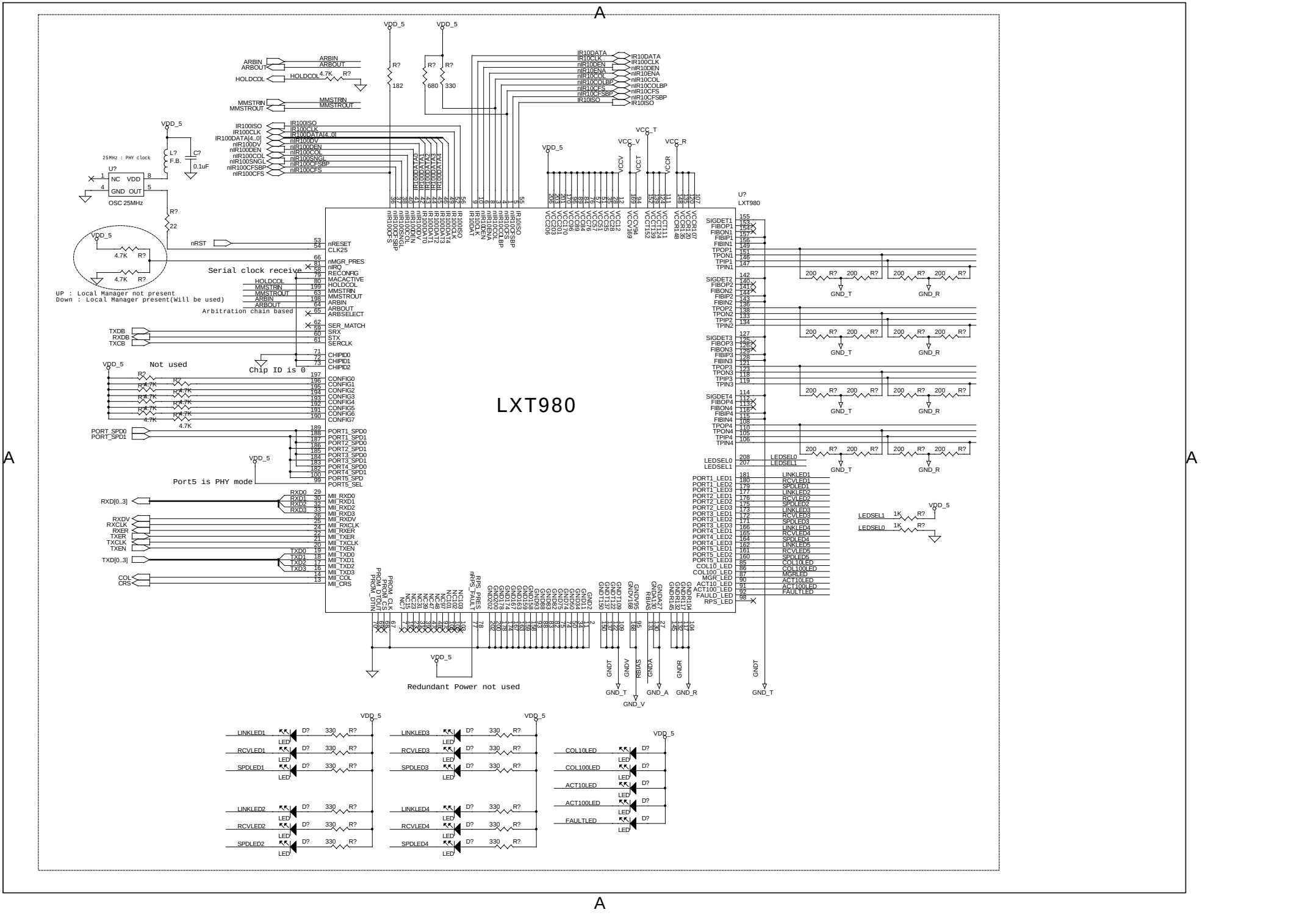
Title	ARM7S BOARD	
Size B	Document Number 1	Rev 0.0
Date: Friday, April 09, 1999	Sheet 1 of 1	



Title	ARM7S BOARD	
Size B	Document Number 1	Rev 0.0
Date: Saturday, April 10, 1999	Sheet 1 of 1	



Title	ARM7S BOARD	
Size: B	Document Number: 1	Rev: 0.0
Date: Friday, April 09, 1999	Sheet 1 of 1	


LXT980
Serial clock receive
Arbitration chain based
Chip ID is 0
Not used
Port5 is PHY mode
Redundant Power not used
UP : Local Manager not present
Down : Local Manager present(Will be used)
OSC 25MHz
25MHz : PHY clock

1 ABOUT SNDS100 BOARD

SYSTEM OVERVIEW

SNDS100(Samsung NetARM Development System for KS32C50100/KS32C5000(A) is a platform that is suitable for code development of SAMSUNG's KS32C50100 16/32-bit RISC microcontroller(NetARM-II) for Ethernet-based system. Also it supports a development of KS32C5000(A)(NetARM-I) in a similar way as SNDS do.

KS32C50100/KS32C5000(A) consists of 16-/32-bit RISC(ARM7TDMI) CPU core, 8-Kbyte unified cache/SRAM, I2C-bus controller, Ethernet controller with 2-channel buffered DMA, 2 HDLC with 4-channel buffered DMA, 2-channel GDMA, 2 UARTs, two 32-bit timers, 18 programmable I/O ports, interrupt controller, and a system manager. it also supports JTAG boundary scan for the application system testing.

SNDS100 consists of KS32C50100/KS32C5000(A) , boot EEPROM(Flash ROM), DRAM module, SDRAM, serial ports for console, two serial communication ports, ethernet interface, configuration switches, and status LEDs/LCD. The Ethernet interface has a complete IEEE802.3 physical layer interface with ethernet hub/router side RJ45 connector configuration.

SNDS100 BOARD OVERVIEW

The SNDS100 shows the basic system-based hardware design which uses the KS32C50100/KS32C5000(A) . It can evaluate the basic operations of the KS32C50100/KS32C5000(A) and develop codes for it as well.

When the KS32C50100/KS32C5000(A) is contained in the SNDS100 , 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 KS32C50100/ KS32C5000(A) with embeddedICETM capability can be debugged directly using the EmbeddedICE Interface.

The SNDS100 function blocks are shown in Figure 1-1.

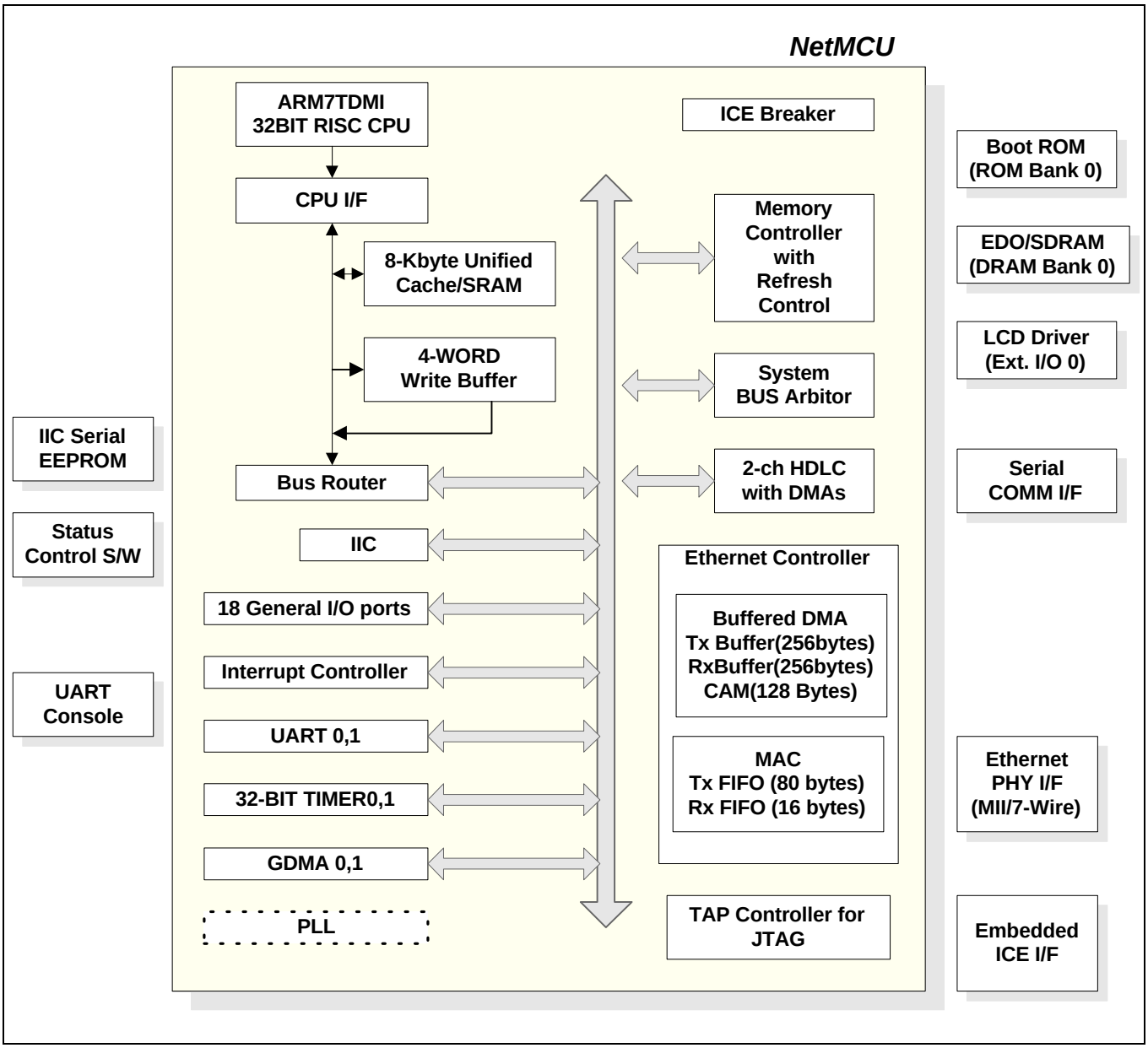


Figure 1-1. SNDS100 Block Diagram

FEATURES

— KS32C50100/KS32C5000(A) : 16/32-bit RISC microcontroller
— Boot ROM : 512K bit, 1M bit, 4M bit, support byte, half-word, word size boot ROM
— DRAM : 72-pin SIMM module with two banks and EDO DRAM support
— SDRAM : Two 4Mx16 with 2banks SDRAM support
— External I/O : status LCD driver
— General I/O : control switches and status display LED
— Two serial ports, one for console
— I2C-bus EEPROM
— Two-channel serial communication interface
— 10/100Mbps Ethernet interface
— EmbeddedICE™ Interface

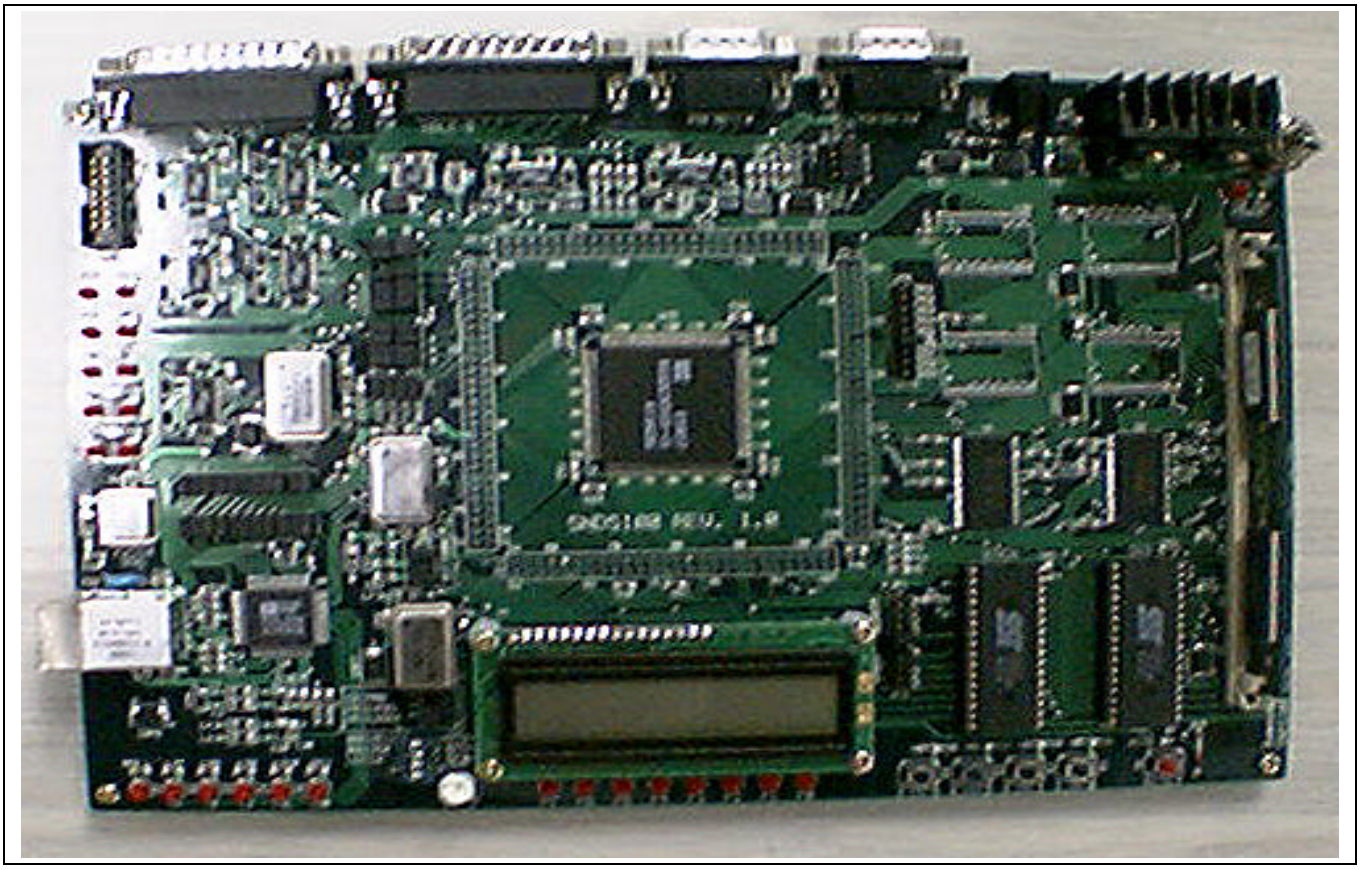


Figure 1-2. SNDS100 Rev1.0 Evaluation Board

CIRCUIT DESCRIPTION

SNDS100 board consists of logic components, several control/status display block, and a debug interface block. SNDS100 board's detailed block diagram, and its components are shown in figure 1-4. SNDS100 board schematics are inserted at the end of this section.

POWER SUPPLY

SNDS100 is designed to operate at 3.3V and 5V. Power to the SNDS100 is supplied through a DC jack power adapter which supports the voltage between 6V and 9V and drives the current at least 850 mA .

SNDS100 board has distributed power plane, with power going separately to the MCU and the main power plane. In case of KS32C5000(A), main power and MCU power has 5V level. But with KS32C50100,MCU power has 3.3V. For this reason, power jumpers J5,J6,J7 and J8 are inserted (see Figure 1-3) .

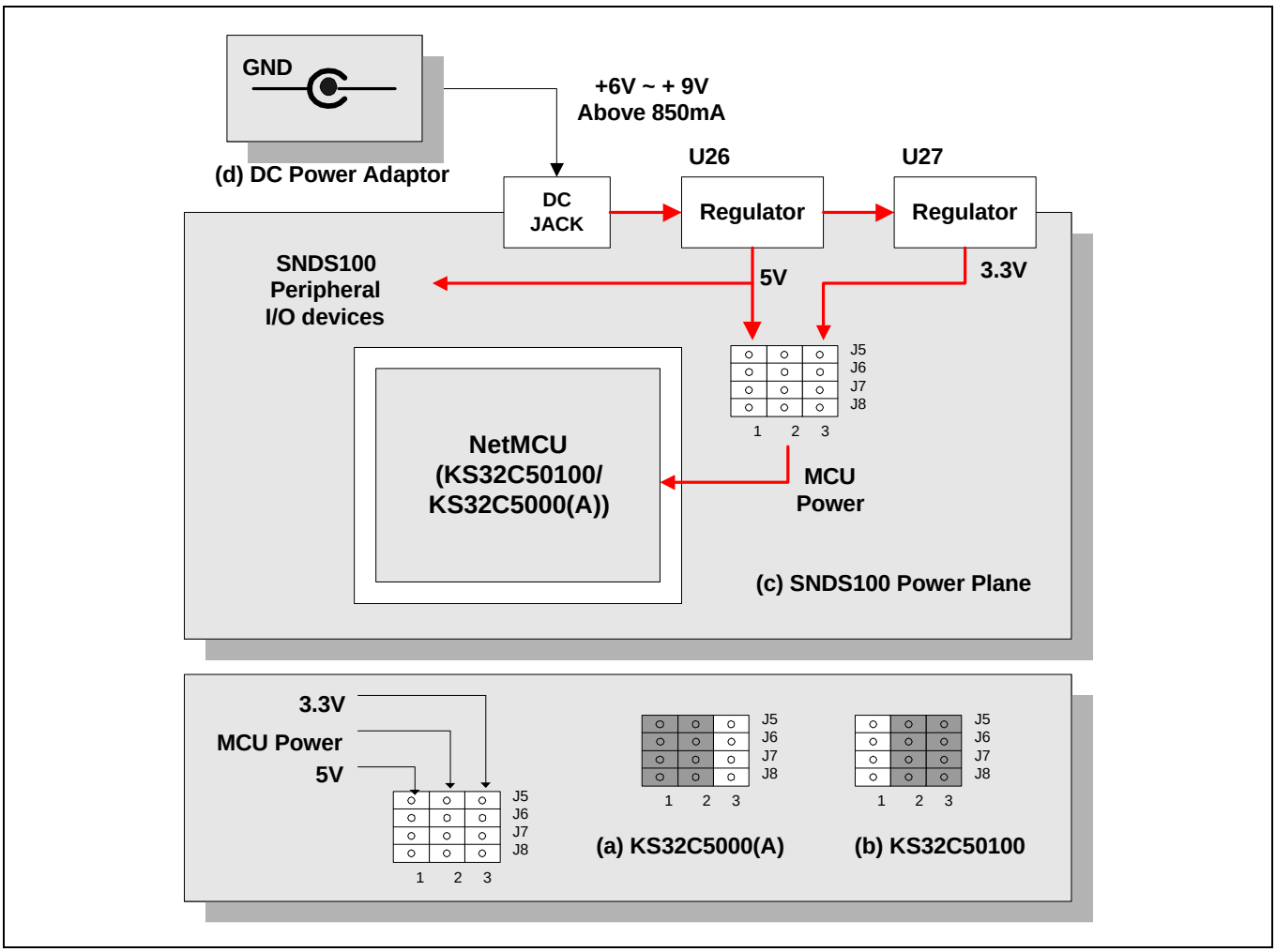


Figure 1-3. SNDS100 Power Plane

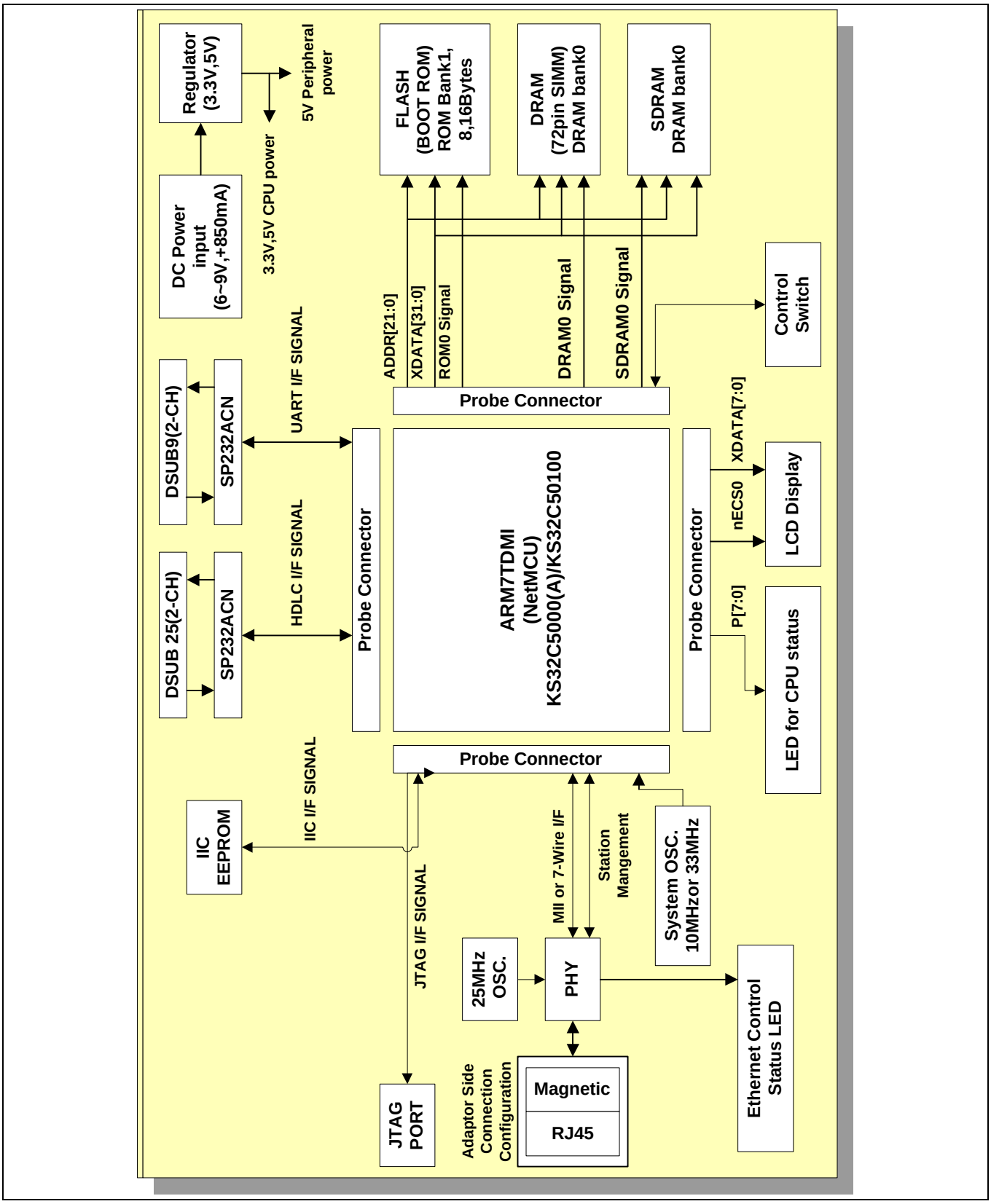


Figure 1-4. Detailed SNDS100 Board Diagram

CLOCK SOURCE AND DISTRIBUTION

The Following clock sources are supported at SNDS100 target board.

System Clock (MCLK)

In case of that the attached device on SNDS100 is KS32C5000(A), then you can use 33MHz oscillator for system clock source. If KS32C50100 device is attached on it, then you can directly assign the 50MHz oscillator to the MCLK input pin or 10MHz with PLL enabled. (See, Figure 1-5)

System Clock Out (MCLKO)

MCLKO is the same signal as internal system clock(MCLK) of KS32C50100/5000(A). This clock can be monitored at MCLKO pin as to assign high to CLKOEN(MCKO clock output enable/disable input) pin . If you want to use SDRAM with KS32C50100, MCLKO should be used. (See, Figure 1-5)

Table 1-1. System clock configurations

DEVICE	JUMPER	1-2(HIGH)	2-3(LOW)	NOTE
KS32C50100/ KS32C5000(A)	TMOD(J2)	(Don' t use)	Normal	Should be set to LOW for normal operation.
KS32C5000(A)	CLKSEL(J3)	MCLK/2	MCLK	SYSTEM Clock(MCLK) select input pin.
KS32C50100		MCLK	PLL output	MCLK input frequency can be controlled by this pin.
KS32C50100/ KS32C5000(A)	CLKOEN(J4)	MCLKO enabled	MCLKO disabled	MCLKO have to be enabled to use SDRAM(only for KS32C50100)

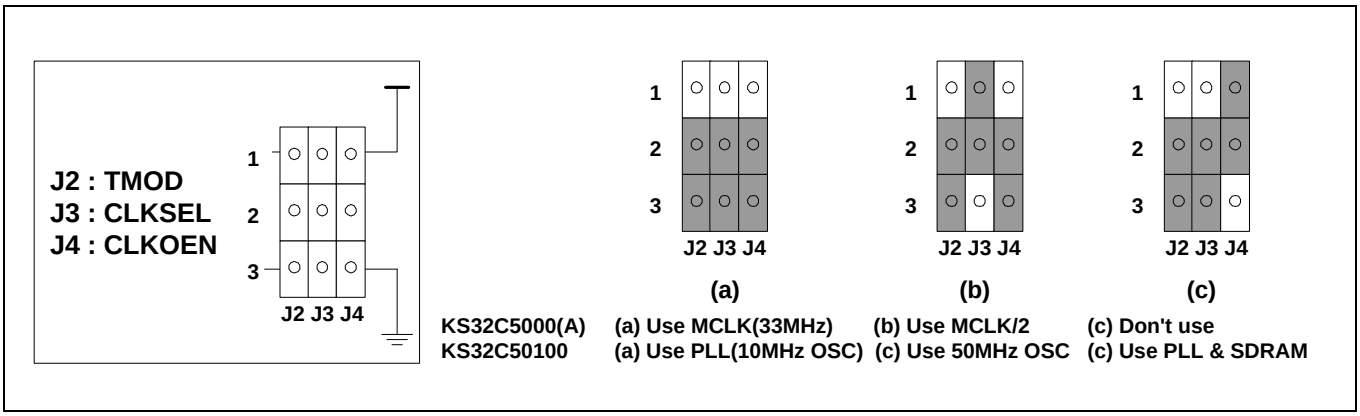


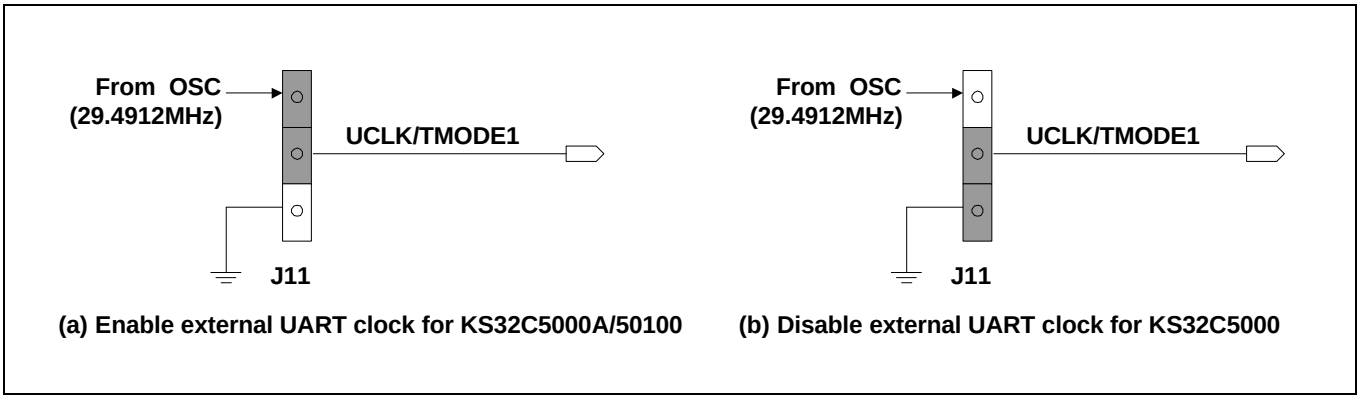
Figure 1-5. The Examples of System Clock(MCLK) Configurations

Ethernet Control Clock

25MHz crystal oscillator have to be used for 100/10Mbps Ethernet PHY control clock.

External UART Clock

KS32C50100/5000A support the External UART Clock input pin (UCLK[64]). If you using KS32C5000 with SNDS100, This UCLK input pin have to be assigned to LOW. Because it is used as the test mode selection pin(TMOD1) for KS32C5000. (see Figure 1-6).



(a) Enable external UART clock for KS32C5000A/50100 **(b) Disable external UART clock for KS32C5000**

Figure 1-6. External UART Clock Configurations

RESET LOGIC

The nRESET(System Reset Signal) must be held to low level at least 540 master clock cycles to reset a KS32C50100/5000(A). nRESET and nTRST(JTAG Reset signal) is logic anded. But, if you want to use circuit emulator(ex, Embedded ICE) for 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.

Therefore, the current SNDS100 Rev.1.0 schematic for reset logic have to be updated. It is referred to “Section 4. JTAG for Embedded ICE Interface” .

SNDS100 SYSTEM CONFIGURATIONS

SNDS100 board provides Big-/Little- endian mode with KS32C50100/5000A and supports Byte/Halfword/Word access the data bus.

Are you using KS32C5000(A) on SNDS100 board?. Then you have to pull-out the all jumper from J9. Because of these pins are CPU monitoring pins(CPUMP[2:0]) of KS32C5000(A). But, in case of KS32C50100, these pin functions are changed to the filter input and analog power for the internal PLL circuit. See the schematic file which is at the end of this section.

Flash Boot ROM

DIP type ROM sockets(U17, U18) are on SNDS100 for to support the byte(8 bits) or halfword(16 bits) BOOT ROM even though the data bus for ROM Bank0 can be configured by B0SIZE[1:0] pins up to 32bit.

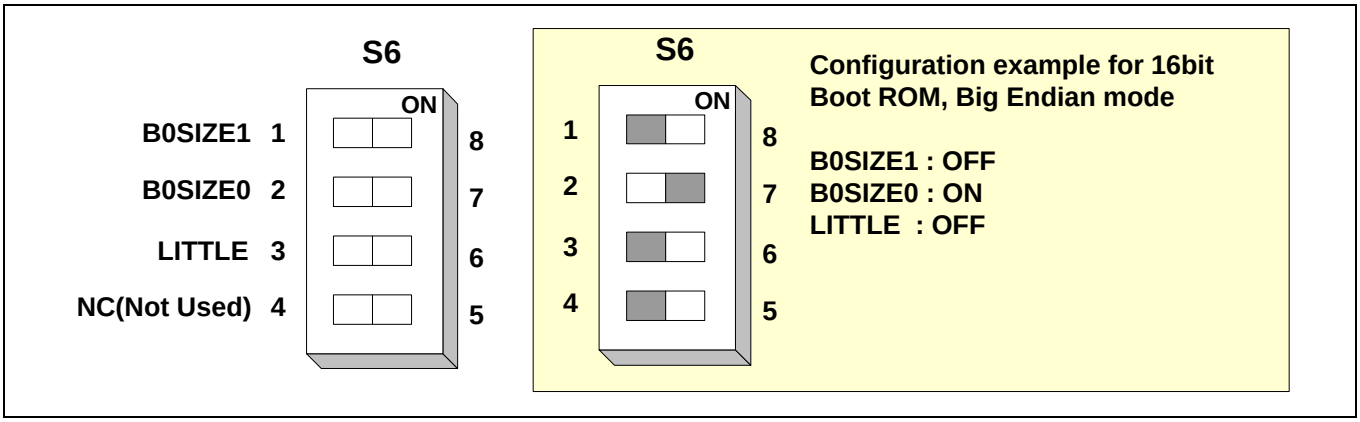


Figure 1-7. SNDS100 Board Configurations

Table 1-2. ROM Bank0 Data Bus Width

PIN FUNCTIONS	S6: B0SIZE[1:0]	PIN VALUE	DESCRIPTIONS
ROM Bank0 Data Bus Width configuration	ON, ON	"00"	Reserved
	ON,OFF	"01"	Byte (8 bits)
	OFF,ON	"10"	Half-word (16 bits)
	OFF,OFF	"11"	Word (32 bits)

Table 1-3. Endian Mode Configuration

PIN FUNCTIONS	S6: LITTLE	PIN VALUE	DESCRIPTIONS
Endian Mode Selection	ON	"1"	Little endian mode (KS32C50100/KS32C5000A)
	OFF	"0"	Big endian mode. KS32C5000 is fixed to this mode.

DRAM/SDRAM Configurations

SNDS100 has the 72-pin SIMM module on the board for one bank DRAM. KS32C50100 can support Synchronous DRAM(SDRAM). In this case, SDRAM or DRAM memory can be selected alternatively using by SYSCFG register. KS32C5000(A) did not support SDRAM type. Using these device with SNDS100, JP2 have to be set to select DRAM.

Bank select jumper for DRAM/SDRAM, JP2/JP1 on SNDS100 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 SNDS100 , 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.

ROM and External I/O Bank Chip Select Jumpers(J8)

SNDS100 also provides ROM and External I/O bank selection jumper for the purpose of each bank test using by SRAM(U20, U21, U19, U22).

— RS1 ~ R25 : ROM/SRAM/FLASH bank selection jumpers.

— ES0 ~ ES3 : External I/O bank selection jumpers.

These bank selection jumper(J8) have to be enabled only one bank, if you want to use it.

STATUS LCD DRIVER

SNDS100 provide LCD display to indicate SNDS100 status. External I/O bank 0 is used to control the LCD driver. The LCD Driver interface connector pin numbers are described in Table 1-4.

Table 1-4. LCD Driver Interface

Pin No.	Descriptions
1	GND
2	VCC
3	Resolution control
4	A[1]
5	A[0]
6	Chip select
[14:7]	DATA[7:0]

GENERAL I/O PORTS

KS32C50100/KS32C5000(A) 's general I/O ports are used for SNDS100 key interrupt input and LED status display. The function of control switch and the status of LED can be defined by user software.

Table 1-5. General I/O Configurations on SNDS100

General I/O port number	I/O type	Descriptions
P[7:0]	Output	LED display
P[11:8]	Input	Key input pad (External interrupt input pins).
P[17:16]	Output	HDLC Data Set Ready Signal output(nDSRB/nDSRA).

ETHERNET INTERFACE

KS32C50100/KS32C5000(A) has one 10-/100-Mbps Ethernet controller. SNDS100 supports 10-/100-Mbps Ethernet interface, A PHY chip set used in SNDS100 is able to operate 10-/100-M bps using auto-negotiation and communicate with KS32C50100/KS32C5000(A) using an MII interface.

SNDS100 Ethernet connector(RJ45) has Ethernet adapter side pin configuration which supports communication between the SNDS100 and the host PC's NIC. You can connect SNDS100 to hub or router direct without twisting cable. Both receive and transmit domains must be connected to the digital domain through a ferrite bead or inductor. The value of the inductor is from 0.1uH to 1uH.

Jumpers(J3-1,J3-2) are located between the MII interface of NetMCU MAC and PHY for another PHY chip. User who would like to use any other vendor' s PHY chip can use this jumpers as interface a daughter board. These jumpers should be always enabled for MAC evaluation.

The PCB power plane and RJ45 connector(J1) configurations are shown in Figure 1-8.

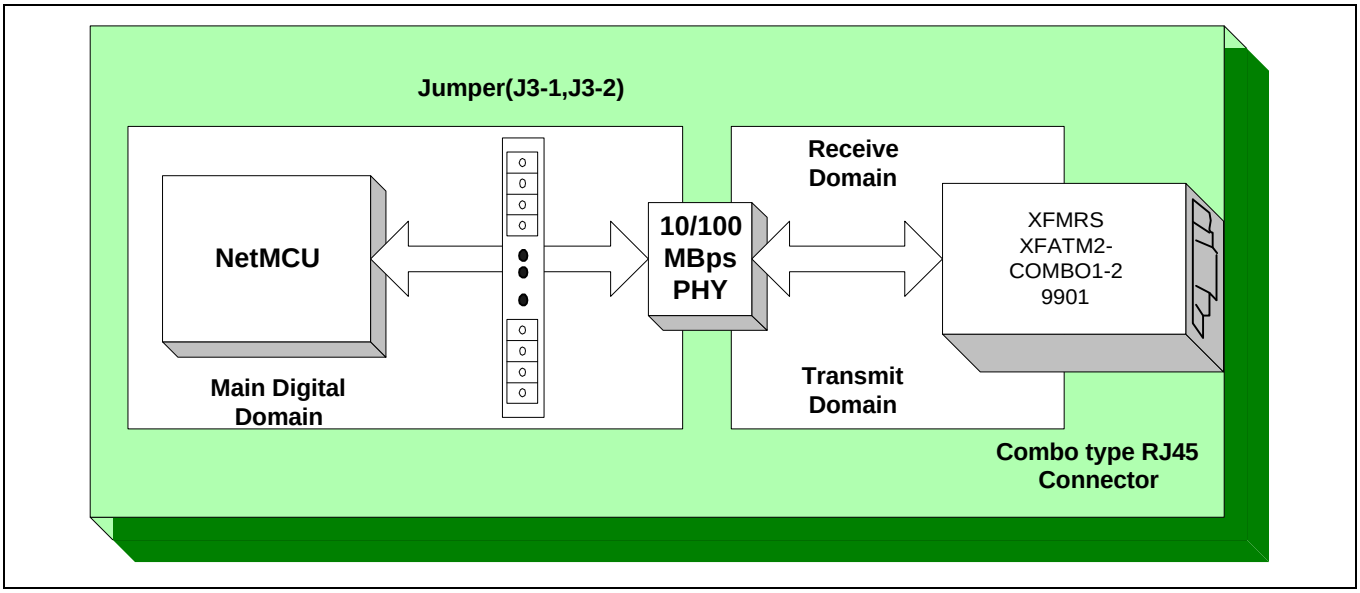


Figure 1-8. Ethernet I/F PCB Power Plane on SNDS100

Table 1-6. RJ45 Pin Configurations for Adapter side

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

Ethernet Status LED

— LED location : D1, D2, D3, D4, D5

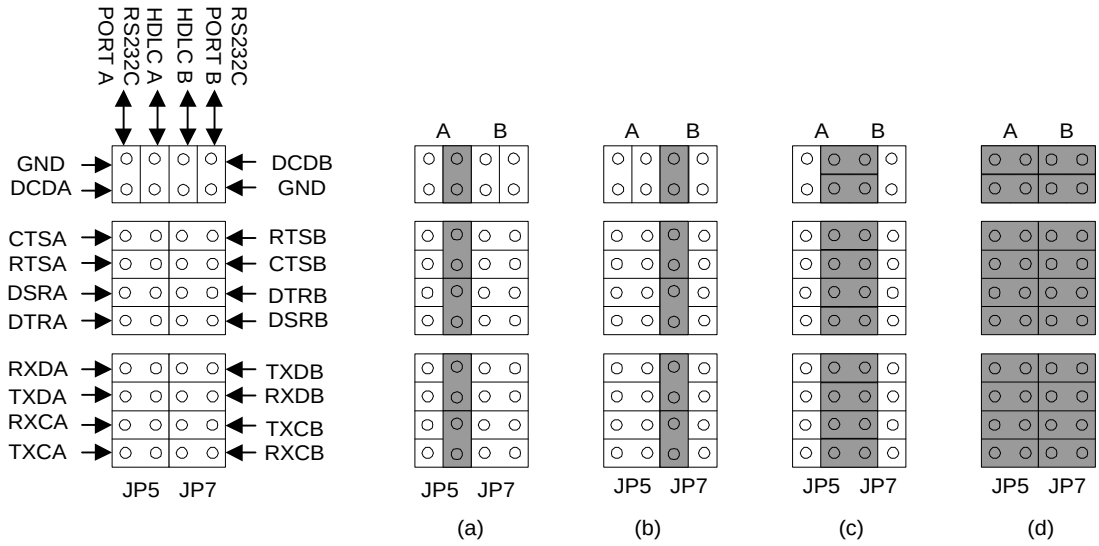
— Indicate the line status, operation mode and Speed.

Table 1-7. Ethernet Status LED.

LED	FUNCTIONS	DESCRIPTIONS
D1	ACTIVITY	Line Status LED. Indicate Full-Duplex operation.
D2	RX	Receive data LED.
D3	TX	Transmit data LED.
D4	LI	Link integrity LED.
D5	COL	Collision detected.
	SPEED	Activity LED. Indicate 100Mbps operation.

HDLC TEST CIRCUIT CONFIGURATIONS

SNDS100 board provides HDLC external loop back test jumper(JP5,JP6,JP7). Configuring the jumper set, you can test the external loop back between the HDLC channel. Also you can select the HDLC Oscillator for external HDLC clock input using by JP6.



(a) HDLC-A channel external loop back test (From channel A to A)
(b) HDLC-B channel external loop back test (From channel B to B)
(c) HDLC channel A to B external loop back test (From channel A to B or B to A)
(d) HDLC channel A to B external test (Interface with V.24/RS232)

Figure 1-9. HDLC Jumper Setting for External Loop Back Test

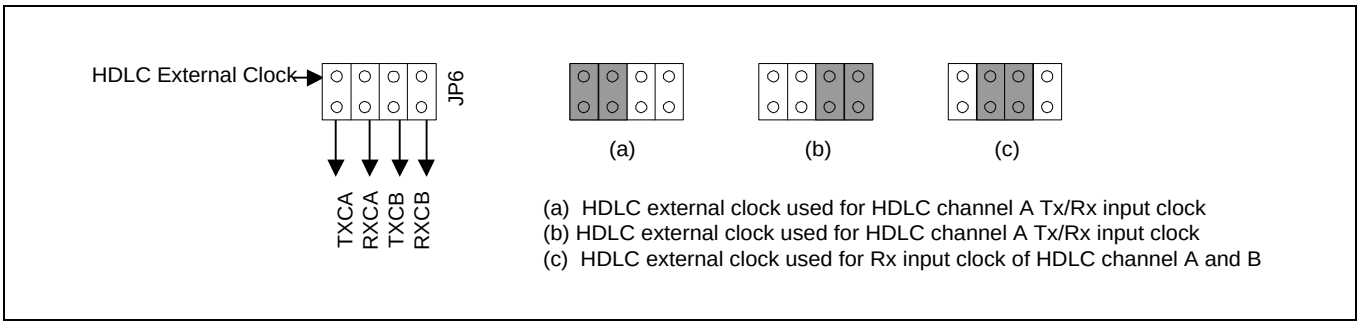


Figure 1-10. HDLC Tx/Rx Clock Input Configurations

Serial (HDLC, UART) & JTAG Interface

SNDS100 board supports the two 9DIP SUB serial connector for two channel UART(P2A SIO-0, P2B SIO-1). If you want to get the connector pin configurations, please refer to "Section 2".

NetMCU device have the two-channel HDLC(High Level Data Link Controllers) for serial communication. The SNDS100 provides serial communication port(HDLC-A P1A, HDLC-B P1B) with V.24/RS-232 interface.

SNDS100 support JTAG port. It can be used as Circuit emulator(ex, Embedded ICE) interface for boundary scan test and debugging channel for application. For the details, See the "Section 4. System design".

SNDS100 REV. 1.0. BOARD SCHEMATICS

1. MAIN.SCH : SNDS100 TOP SCHEMATIC
2. MCU.SCH : NetMCU(KS32C50100/5000(A)) device interface
3. SYSTEM.SCH : REV. 1.0
4. SYSTEM.SCH : REV. 1.1
5. DRAM.SCH : DRAM/SDRAM SCHEMATICS
6. ETHERNET.SCH
7. EXTERNAL.SCH
8. HDLC.SCH : REV. 1.0
9. HDLC.SCH : REV. 1.1
10. ROM.SCH
11. SRAM.SCH
12. UART.SCH : REV. 1.0
13. UART.SCH : REV. 1.1
14. Bill Of Materials

BILL OF MATERIALS OF SNDS100 REV1.0

SNDS100(Samsung NetMCU Development System) Revised: Saturday, January 16, 1999
MAIN.SCH Revision: 1.0

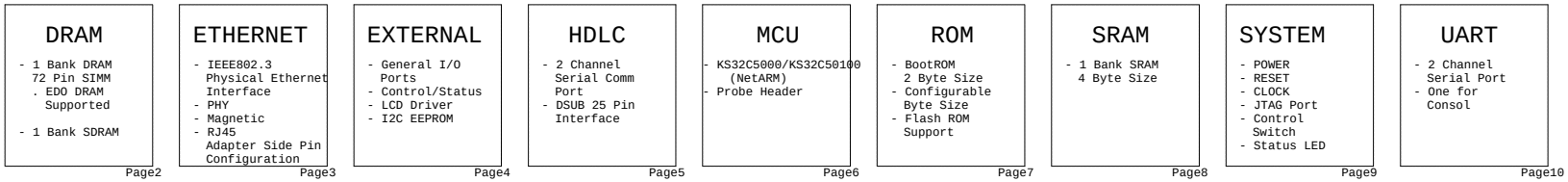
KIMS TECHNOLOGY CO.
#H-305 Konggu sangga ilbeongi
#636-62, Kuro-Dong, Kuro-Gu
Seoul, Korea 152-050
E-mail: kimstek@unitel.co.kr

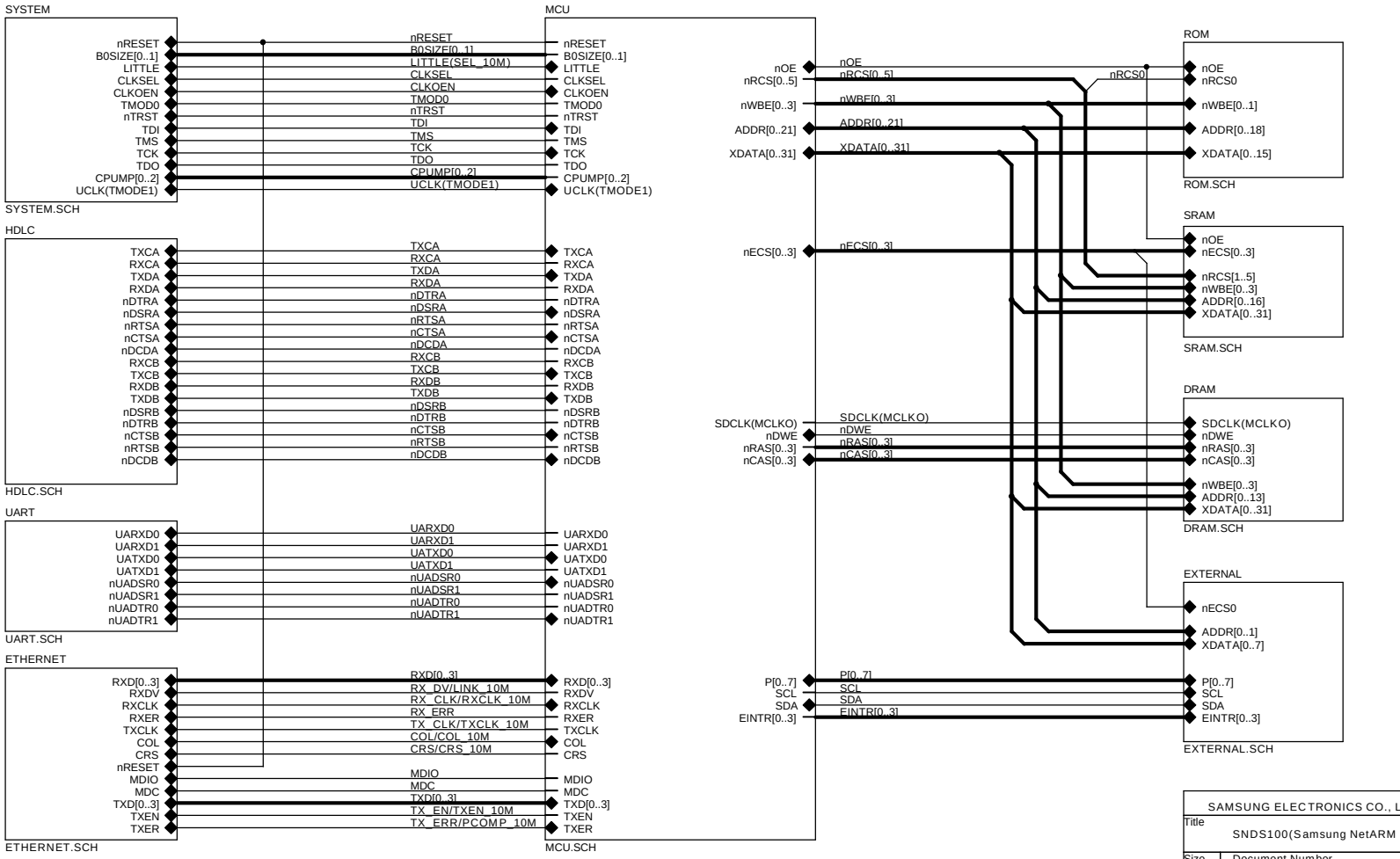
Bill Of Materials March 5,1999 12:22:22 Page1

Item	Quantity	Reference	Part
1	98	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C14, C15, C16, C17, C18, C19, C20, C21, C25, C26, C27, C30, C31, C32, C33, C34, C35, C36, C37, C38, C39, C40, C41, C42, C43, C44, C45, C46, C47, C48, C49, C50, C51, C52, C53, C54, C55, C56, C57, C58, C59, C60, C61, C62, C63, C64, C65, C66, C67, C68, C69, C70, C71, C72, C73, C74, C75, C76, C77, C78, C79, C80, C81, C82, C83, C84, C85, C86, C87, C88, C89, C94, C98, C99, C100, C101, C102, C103, C104, C109, C110, C111, C112, C113, C114	0.1uF
2	1	C22	0.001uF/2KV
3	5	C23, C24, C28, C91, C120	10uF
4	1	C90	10uF/16V
5	1	C92	0.1uF
6	1	C93	820pF
7	1	C95	100uF/6.3V
8	1	C96	100uF/16V
9	1	C97	100uF/10V
10	8	C105, C106, C107, C108, C115, C116, C117, C118	330pF
11	25	D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15, D16, D17, D18, D19, D20, D21, D22, D23, D26, D27	LED
12	1	D24	1N4148
13	1	D25	1N4004
14	3	JP1, JP2, JP6	HEADER 4x2
15	2	JP5, JP7	HEADER 10x2
16	1	JP8	HEADER 9x2
17	1	JP9	HEADER 3x2
18	1	JP10	HEADER 7x2
19	1	J1	XFATM2-COMBO-4
20	8	J2, J3, J4, J5, J6, J7, J8, J11	CON3

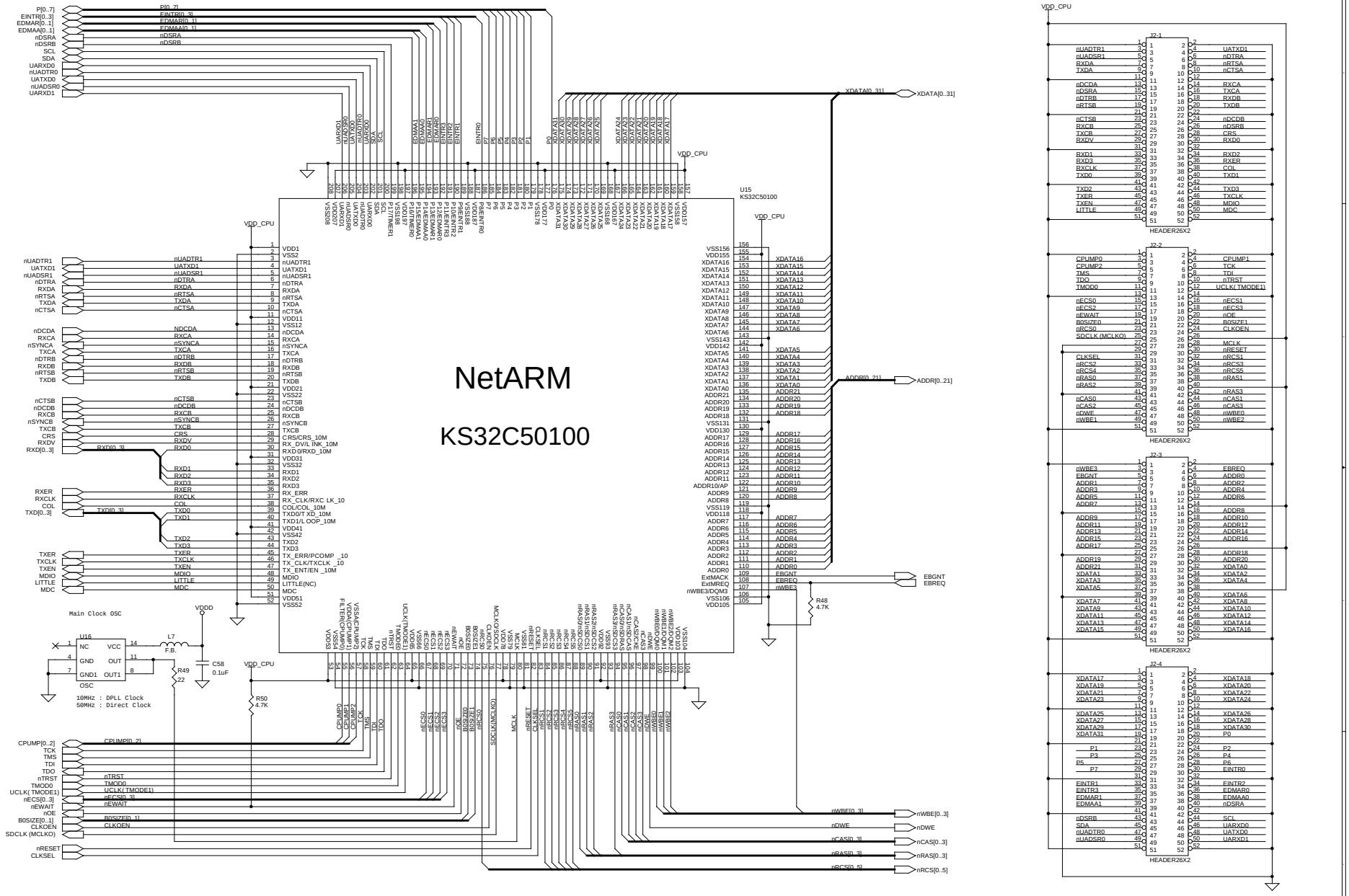
Item	Quantity	Reference	Part
21	2	J9, J12	CON1
22	1	J10	DC_JACK
23	4	J2-1, J2-2, J2-3, J2-4	HEADER 26x2
24	2	J3-1, J3-2	HEADER 11x2
25	9	L1, L2, L3, L4, L5, L6, L7, L9, L11	F.B
26	2	P1A, P1B	CONNECTOR DB25
27	1	P2	CONNECTOR DB9x2
28	12	R1, R2, R3, R4, R5, R6, R21, R47, R49, R51, R57, R61	22
29	16	R7, R8, R9, R10, R11, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R58	680
30	2	R13, R14	49.9
31	1	R15	22K 1%
32	1	R17	100
33	2	R18, R59	1K
34	3	R19, R34, R35	2K
35	4	R20, R28, R52, R60	10K
36	8	R22, R23, R24, R25, R26, R27, R29, R30	330
37	9	R31, R32, R33, R36, R48, R50, R54, R55, R56	4.7K
38	1	R53	33
39	5	S1, S2, S3, S4, S5	SW PUSHBOTTON
40	1	S6	SW_DIP4
41	1	S7	SW ON/OFF
42	2	U1, U2	KM416S4020B
43	1	U3	DRAM_SIMM
44	1	U4	LEVELONE
45	1	U5	OSC(25MHz)
46	2	U6, U23	7414
47	1	U7	LCON14
48	1	U8	KS24C641
49	7	U9, U10, U11, U12, U13, U28, U29	MAX232
50	3	U14, U16, U25	OSC
51	1	U15	KS32C50100
52	2	U17, U18	29E010
53	4	U19, U20, U21, U22	KS681000C_55(SOP)
54	1	U24	7408
55	1	U26	78R05
56	1	U27	78R33

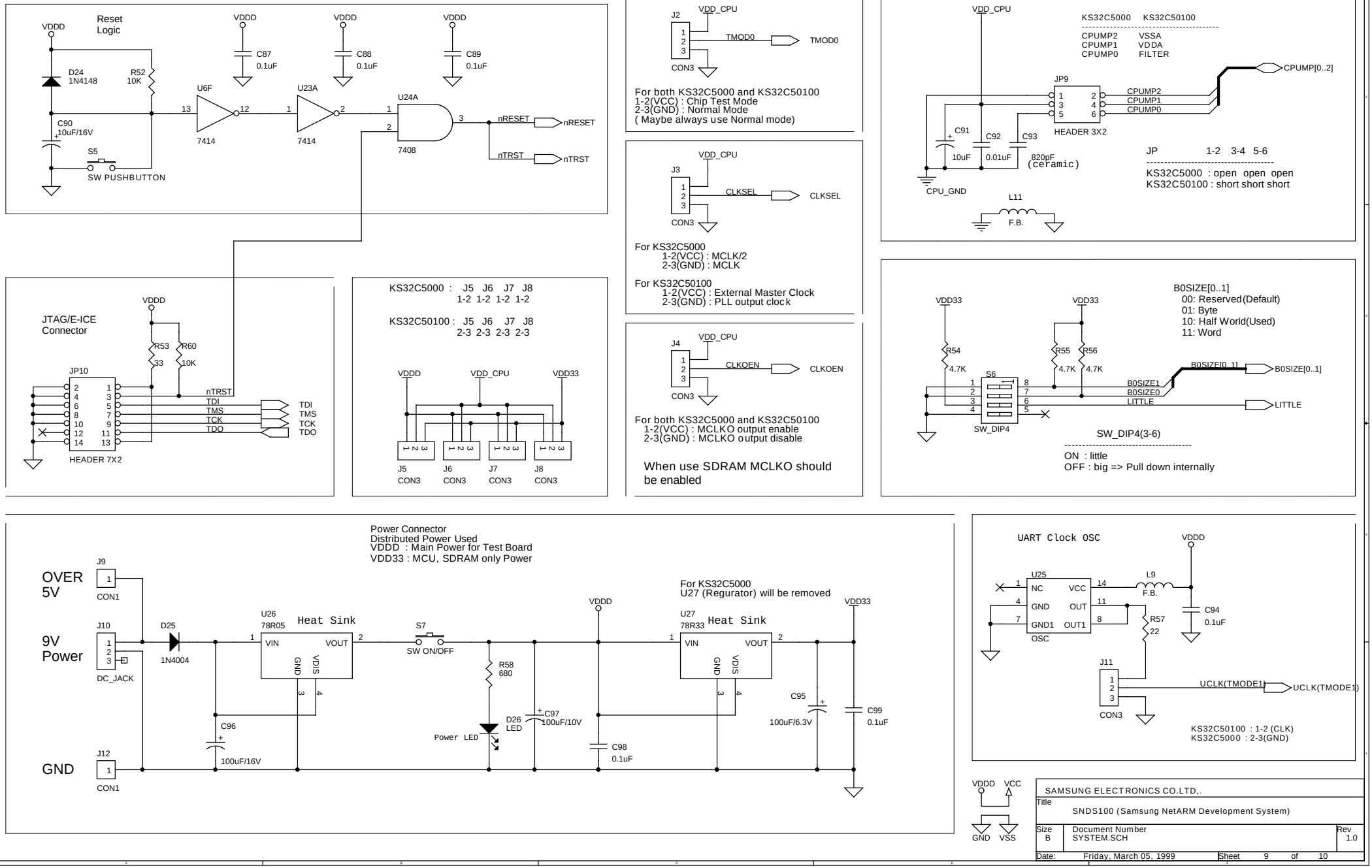
SNDS100 Board Ver1.0 (for NetARM : KS32C5000/KS32C50100)

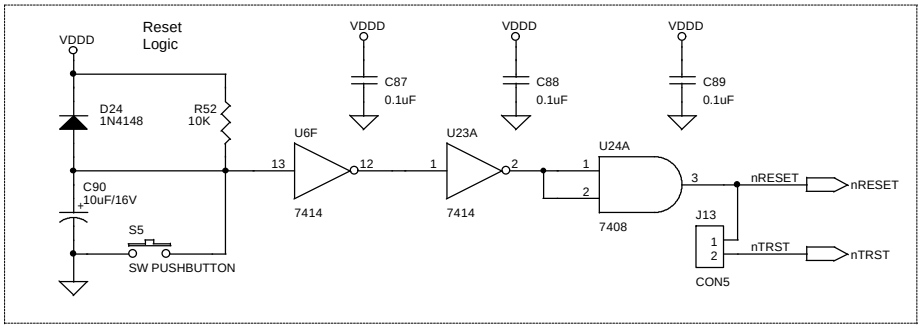


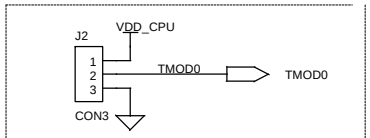


SAMSUNG ELECTRONICS CO., LTD.		
Title: SNDS100(Samsung NetARM Development System)		
Size: B	Document Number: MAIN.SCH	Rev: 1.0
Date: Friday, March 05, 1999	Sheet 1 of 10	

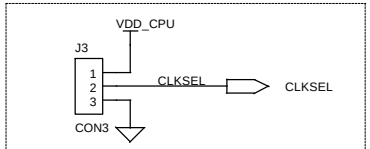

NetARM
KS32C50100
U15
KS32C50100
Main Clock OSC
OSC
10MHz : DPLL Clock
50MHz : Direct Clock
HEADER26X2
J2-1
J2-2
J2-3
J2-4
SAMSUNG ELECTRO NICS CO.LTD.,
SNDS100 (Samsung NetA RM Development System)
MCU.SCH
Saturday, January 1 6, 1999


Reset Logic
VDDD
D24 1N4148
R52 10K
C90 10uF/16V
S5
SW PUSHBUTTON
U6F
U23A
U24A
7414
7414
7408
C87 0.1uF
C88 0.1uF
C89 0.1uF
nRESET
nTRST
JTAG/E-ICE Connector
JP10
R53 33
R60 10K
HEADER 7X2
TDI
TMS
TCK
TDO
KS32C5000 : J5 J6 J7 J8
1-2 1-2 1-2 1-2
KS32C50100 : J5 J6 J7 J8
2-3 2-3 2-3 2-3
VDD_CPU
VDD33
J5 J6 J7 J8
CON3
J2
TMOD0
For both KS32C5000 and KS32C50100
1-2(VCC) : Chip Test Mode
2-3(GND) : Normal Mode
(Maybe always use Normal mode)
J3
CLKSEL
For KS32C5000
1-2(VCC) : MCLK/2
2-3(GND) : MCLK
For KS32C50100
1-2(VCC) : External Master Clock
2-3(GND) : PLL output clock
J4
CLKOEN
For both KS32C5000 and KS32C50100
1-2(VCC) : MCLKO output enable
2-3(GND) : MCLKO output disable
When use SDRAM MCLKO should be enabled
KS32C5000 KS32C50100
CPUMP2 VSSA
CPUMP1 VDDA
CPUMP0 FILTER
JP9
HEADER 3X2
CPUMP[0..2]
C91 10uF
C92 0.01uF
C93 820pF (ceramic)
CPU_GND
L11
F.B.
JP 1-2 3-4 5-6
KS32C5000 : open open open
KS32C50100 : short short short
B0SIZE[0..1]
00: Reserved(Default)
01: Byte
10: Half World(Used)
11: Word
R54 4.7K
R55 4.7K
R56 4.7K
S6
SW_DIP4
B0SIZE1
B0SIZE0
LITTLE
SW_DIP4(3-6)
ON : little
OFF : big => Pull down internally
Power Connector
Distributed Power Used
VDDD : Main Power for Test Board
VDD33 : MCU, SDRAM only Power
OVER 5V
9V Power
GND
J9
J10
J12
CON1
DC_JACK
D25
1N4004
U26
78R05
Heat Sink
VIN
VOUT
GND
VDIS
S7
SW ON/OFF
R58 680
D26 LED
Power LED
C96 100uF/16V
C97 100uF/10V
C98 0.1uF
For KS32C5000
U27 (Regurator) will be removed
U27
78R33
C95 100uF/6.3V
C99 0.1uF
UART Clock OSC
U25
NC
VCC
GND
OUT
GND1
OUT1
OSC
L9
F.B.
C94 0.1uF
R57 22
J11
UCLK(TMODE1)
KS32C50100 : 1-2 (CLK)
KS32C5000 : 2-3(GND)
VDDD VCC
GND VSS
SAMSUNG ELECTRONICS CO.LTD,.
Title
SNDS100 (Samsung NetARM Development System)
Size B
Document Number SYSTEM.SCH
Rev 1.0
Date: Friday, March 05, 1999
Sheet 9 of 10



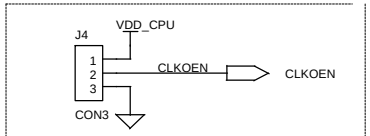


For both KS32C5000 and KS32C50100
1-2(VCC) : Chip Test Mode
2-3(GND) : Normal Mode
(Maybe always use Normal mode)



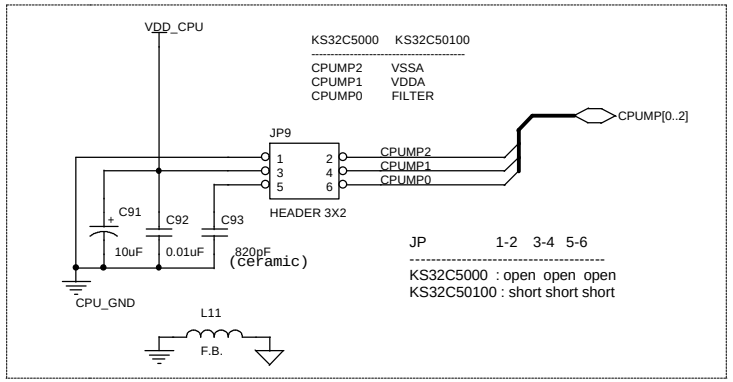
For KS32C5000
1-2(VCC) : MCLK/2
2-3(GND) : MCLK

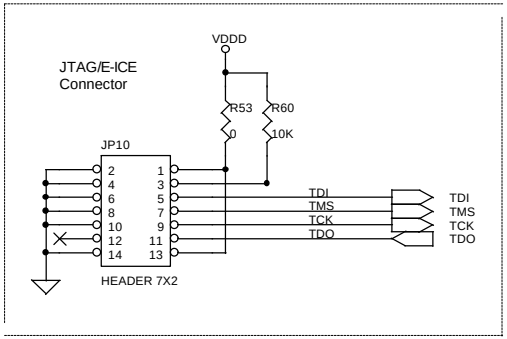
For KS32C50100
1-2(VCC) : External Master Clock
2-3(GND) : PLL output clock



For both KS32C5000 and KS32C50100
1-2(VCC) : MCLKO output enable
2-3(GND) : MCLKO output disable

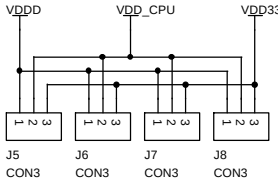
When use SDRAM MCLKO should be enabled

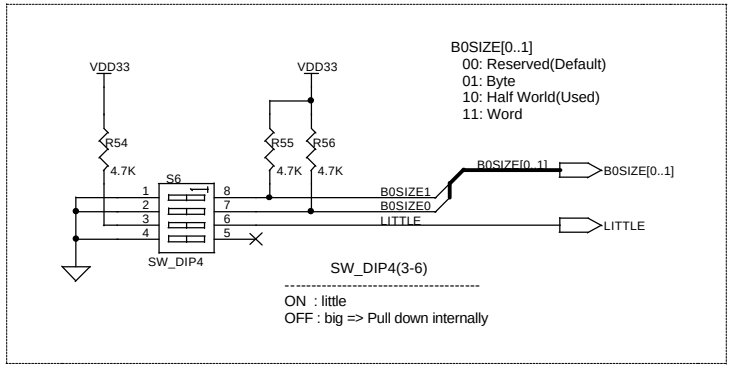


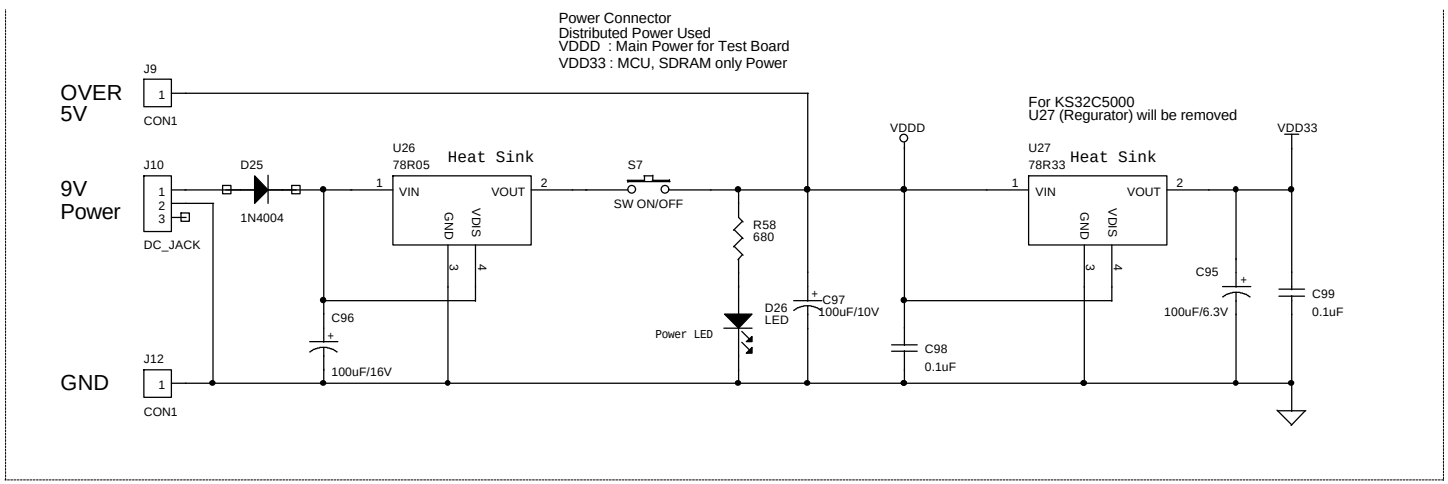


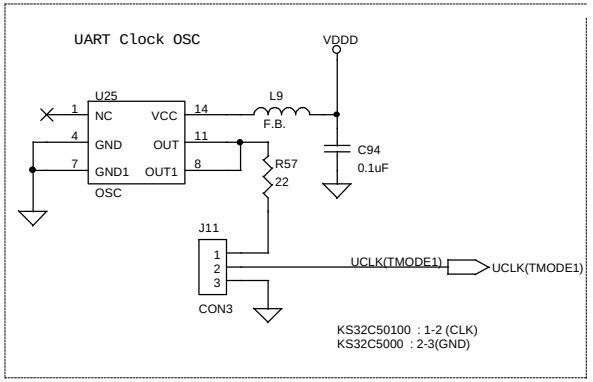
KS32C5000 : J5 J6 J7 J8
1-2 1-2 1-2 1-2

KS32C50100 : J5 J6 J7 J8
2-3 2-3 2-3 2-3

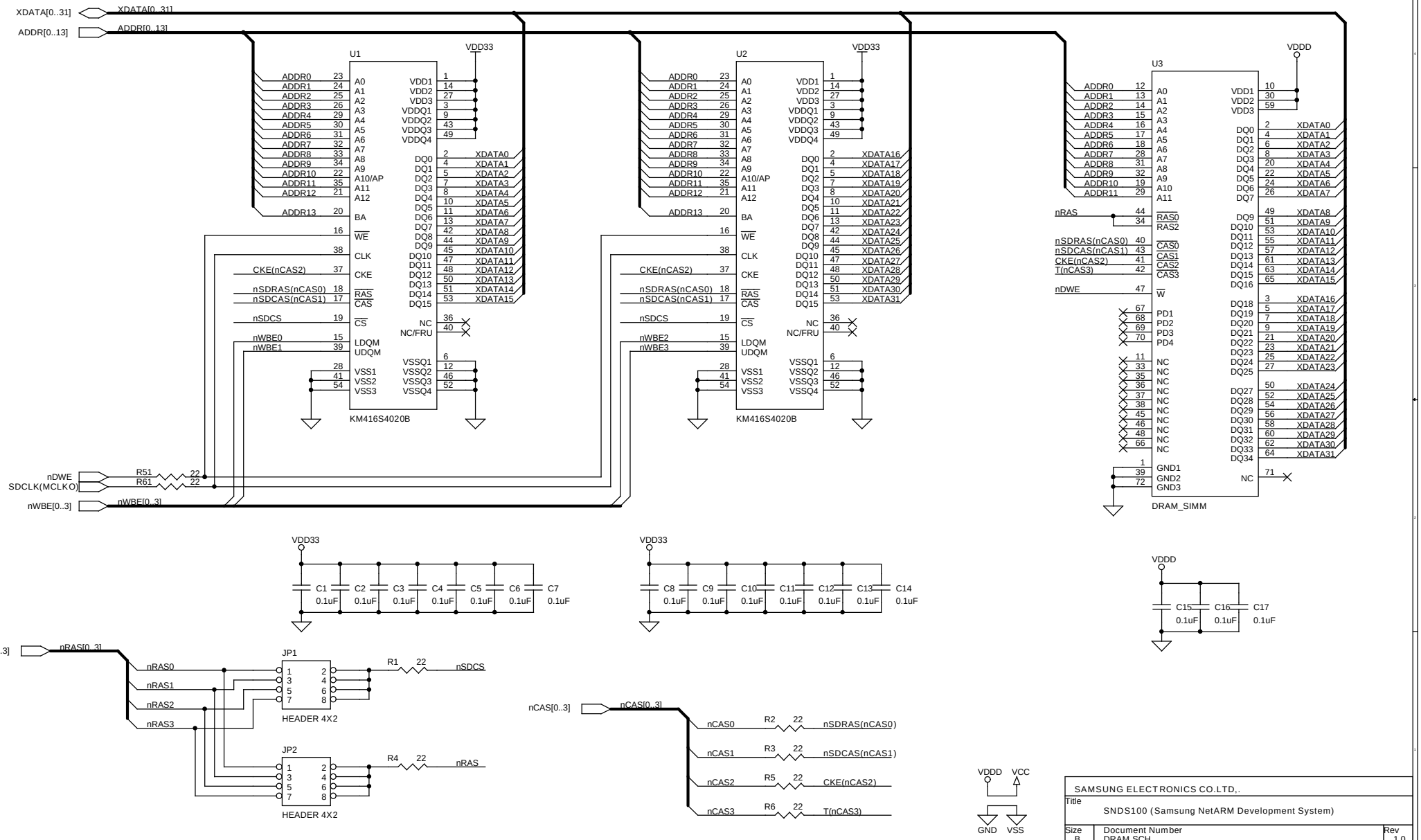


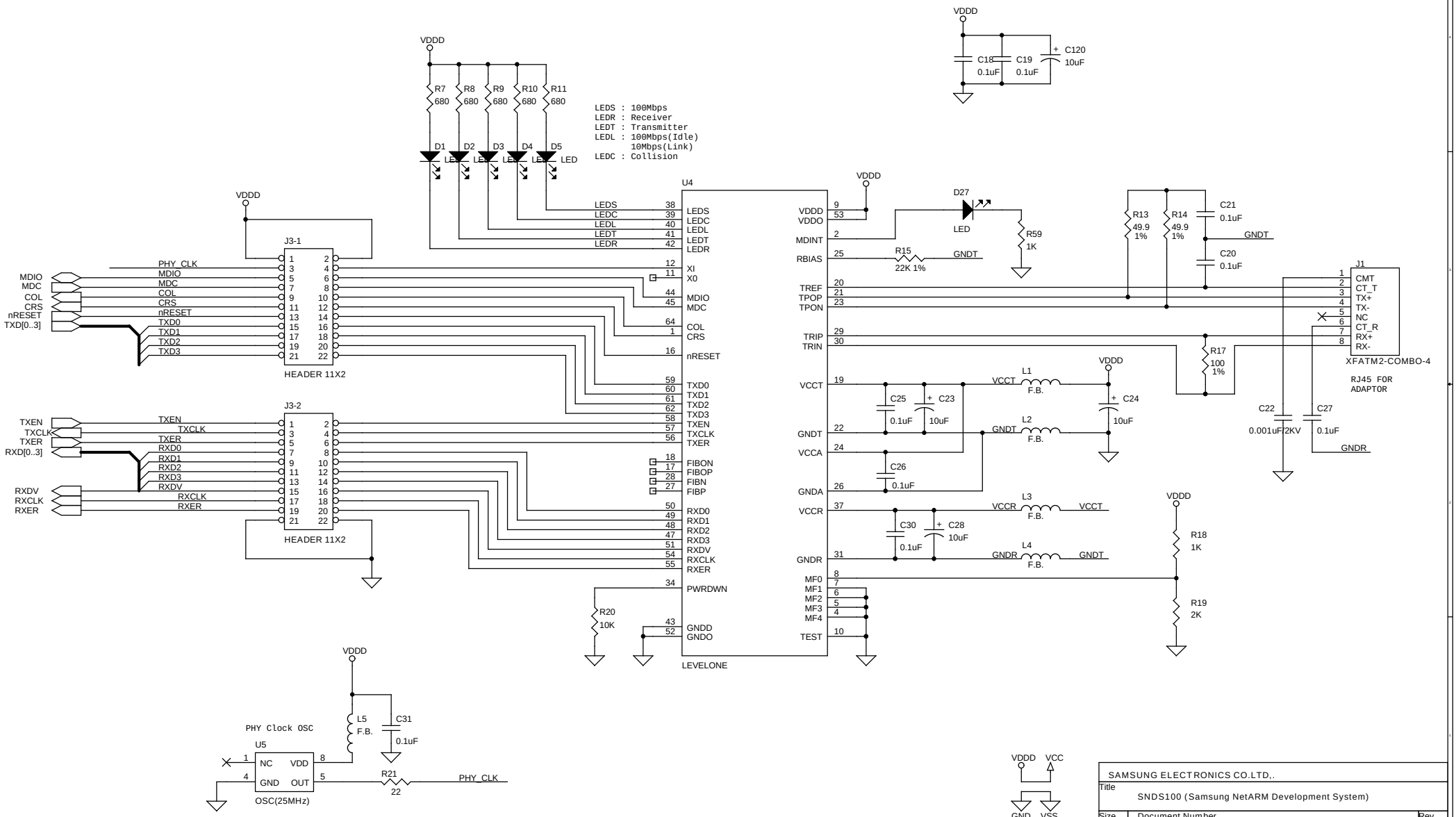


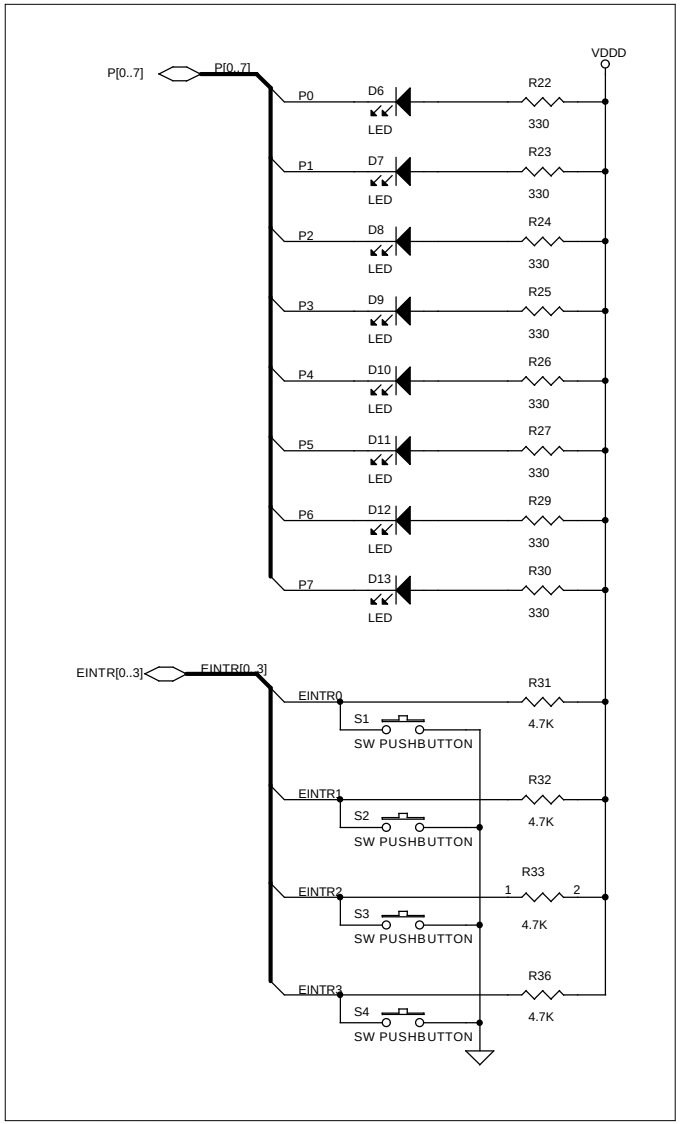


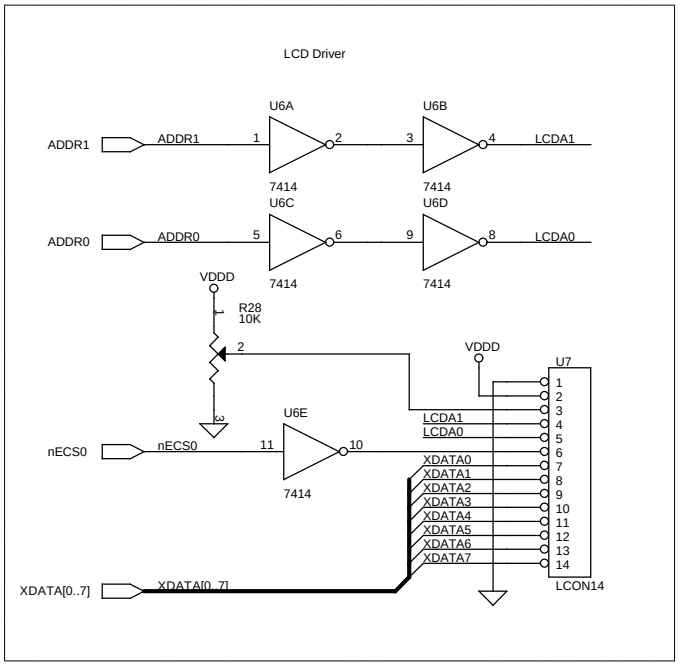


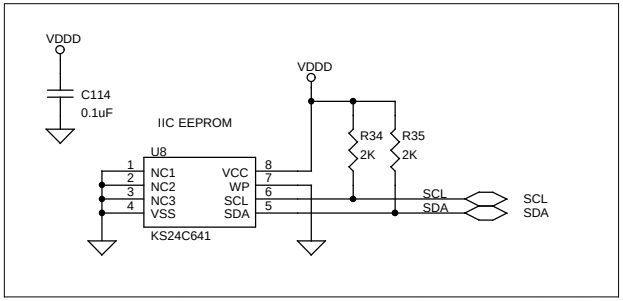
SAMSUNG ELECTRONICS CO.LTD,.		
Title: SNDS100 (Samsung NetARM Development System)		
Size: B	Document Number: SYSTEM.SCH	Rev: 1.1
Date: Thursday, June 10, 1999	Sheet 9 of 10	

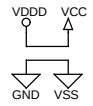

XDATA[0..31]
ADDR[0..13]
U1
U2
U3
VDD33
VDDD
KM416S4020B
DRAM_SIMM
nDWE
SDCLK(MCLKO)
nWBE[0..3]
R51 22
R61 22
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15 C16 C17
0.1uF
nRAS[0..3]
nCAS[0..3]
JP1
JP2
HEADER 4X2
R1 22 nSDCS
R4 22 nRAS
R2 22 nSDRAS(nCAS0)
R3 22 nSDCAS(nCAS1)
R5 22 CKE(nCAS2)
R6 22 T(nCAS3)
VDDD VCC
GND VSS
SAMSUNG ELECTRONICS CO.LTD,.
SNDS100 (Samsung NetARM Development System)
Size B
Document Number DRAM.SCH
Rev 1.0
Date: Thursday, January 21, 1999
Sheet 2 of 10


VDDD
R7 680
R8 680
R9 680
R10 680
R11 680
D1 LED
D2 LED
D3 LED
D4 LED
D5 LED
LEDS : 100Mbps
LEDR : Receiver
LEDT : Transmitter
LEDL : 100Mbps(Idle)
10Mbps(Link)
LEDC : Collision
C18 0.1uF
C19 0.1uF
C120 10uF
U4
LEDS 38
LEDC 39
LEDL 40
LEDT 41
LEDR 42
XI 12
X0 11
MDIO 44
MDC 45
COL 64
CRS 1
nRESET 16
TXD0 59
TXD1 60
TXD2 61
TXD3 62
TXEN 58
TXCLK 57
TXER 56
FIBON 18
FIBOP 17
FIBN 28
FIBP 27
RXD0 50
RXD1 49
RXD2 48
RXD3 47
RXDV 51
RXCLK 54
RXER 55
PWRDWN 34
GNDD 43
GNDO 52
LEVELONE
VDDD 9
VDDO 53
MDINT 2
RBIAS 25
TREF 20
TPOP 21
TPON 23
TRIP 29
TRIN 30
VCCT 19
GNDT 22
VCCA 24
GNDA 26
VCCR 37
GNDR 31
MF0 8
MF1 7
MF2 6
MF3 5
MF4 4
TEST 10
D27 LED
R59 1K
R15 22K 1%
GNDT
R13 49.9 1%
R14 49.9 1%
C21 0.1uF
C20 0.1uF
J1
CMT 1
CT_T 2
TX+ 3
TX- 4
NC 5
CT_R 6
RX+ 7
RX- 8
XFATM2-COMBO-4
RJ45 FOR ADAPTOR
R17 100 1%
C22 0.001uF/2KV
C27 0.1uF
GNDR
L1 F.B.
L2 F.B.
L3 F.B.
L4 F.B.
C25 0.1uF
C23 10uF
C24 10uF
C26 0.1uF
C30 0.1uF
C28 10uF
R18 1K
R19 2K
R20 10K
J3-1
PHY_CLK
MDIO
MDC
COL
CRS
nRESET
TXD0
TXD1
TXD2
TXD3
HEADER 11X2
J3-2
TXEN
TXCLK
TXER
RXD0
RXD1
RXD2
RXD3
RXDV
RXCLK
RXER
HEADER 11X2
PHY Clock OSC
U5
NC
VDD
GND
OUT
OSC(25MHz)
L5 F.B.
C31 0.1uF
R21 22
PHY_CLK
VDDD VCC
GND VSS
SAMSUNG ELECTRONICS CO.LTD,.
Title
SNDS100 (Samsung NetARM Development System)
Size B
Document Number ETHERNET.SCH
Rev 1.0
Date: Saturday, January 23, 1999
Sheet 3 of 10

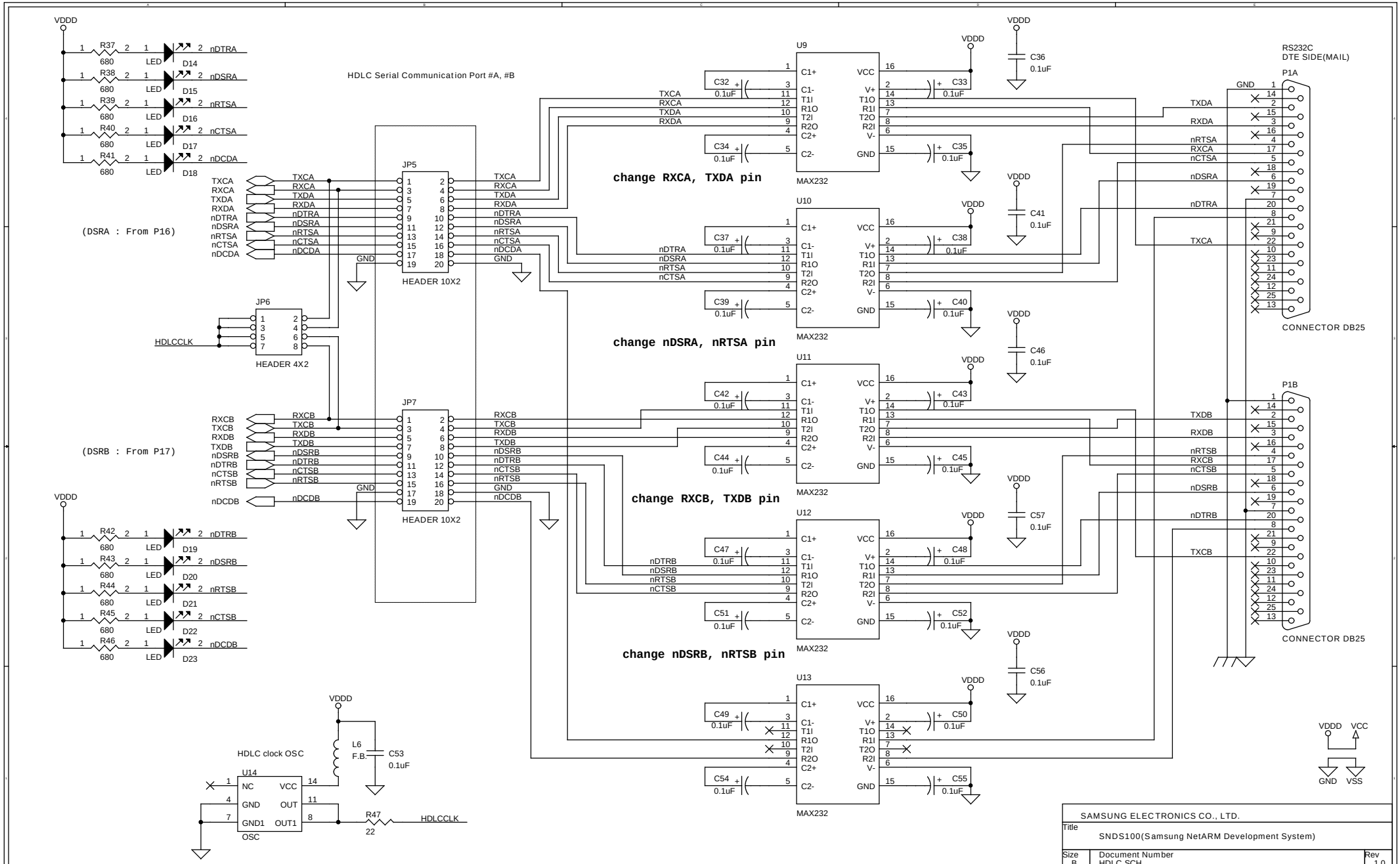


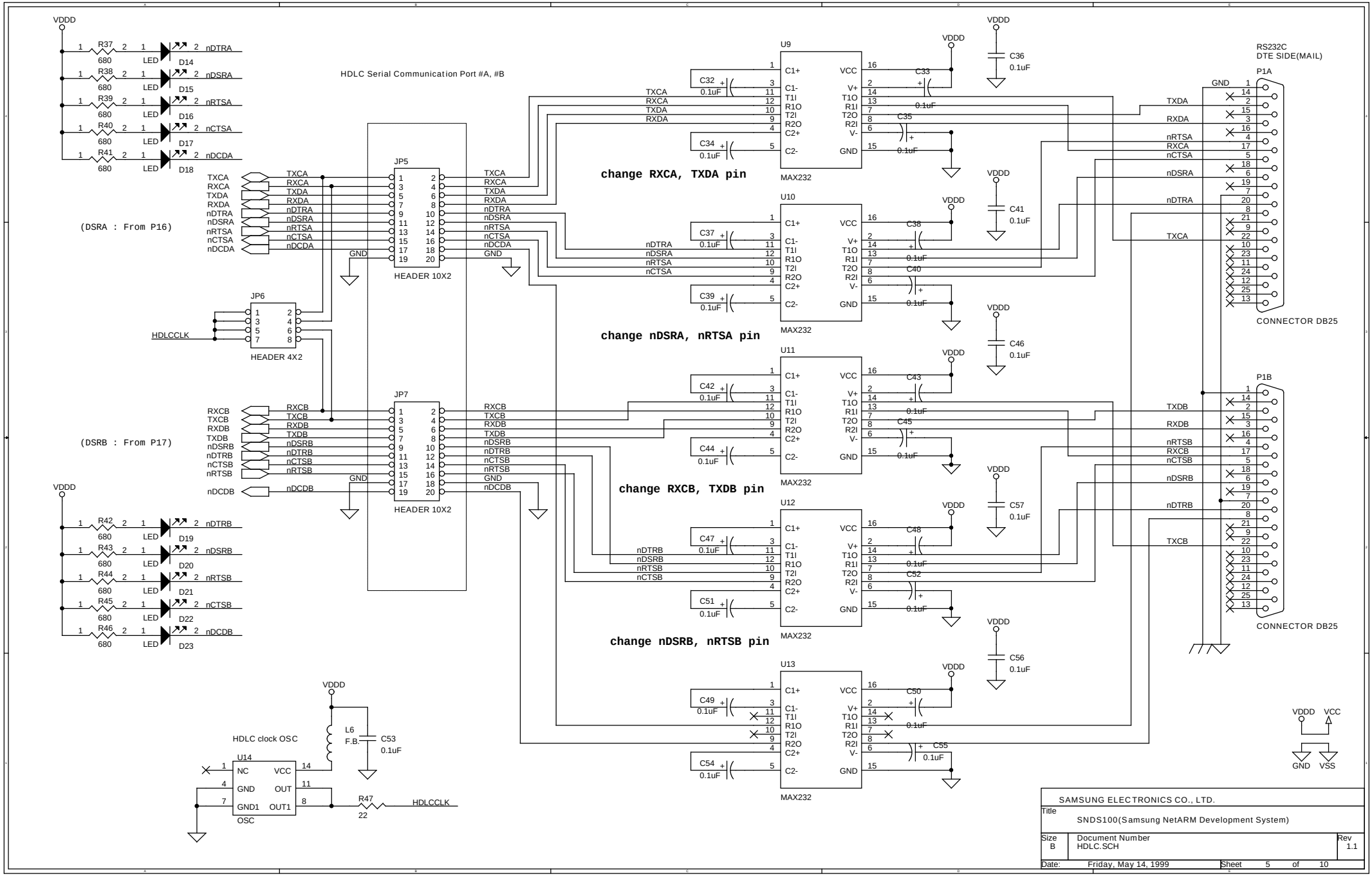




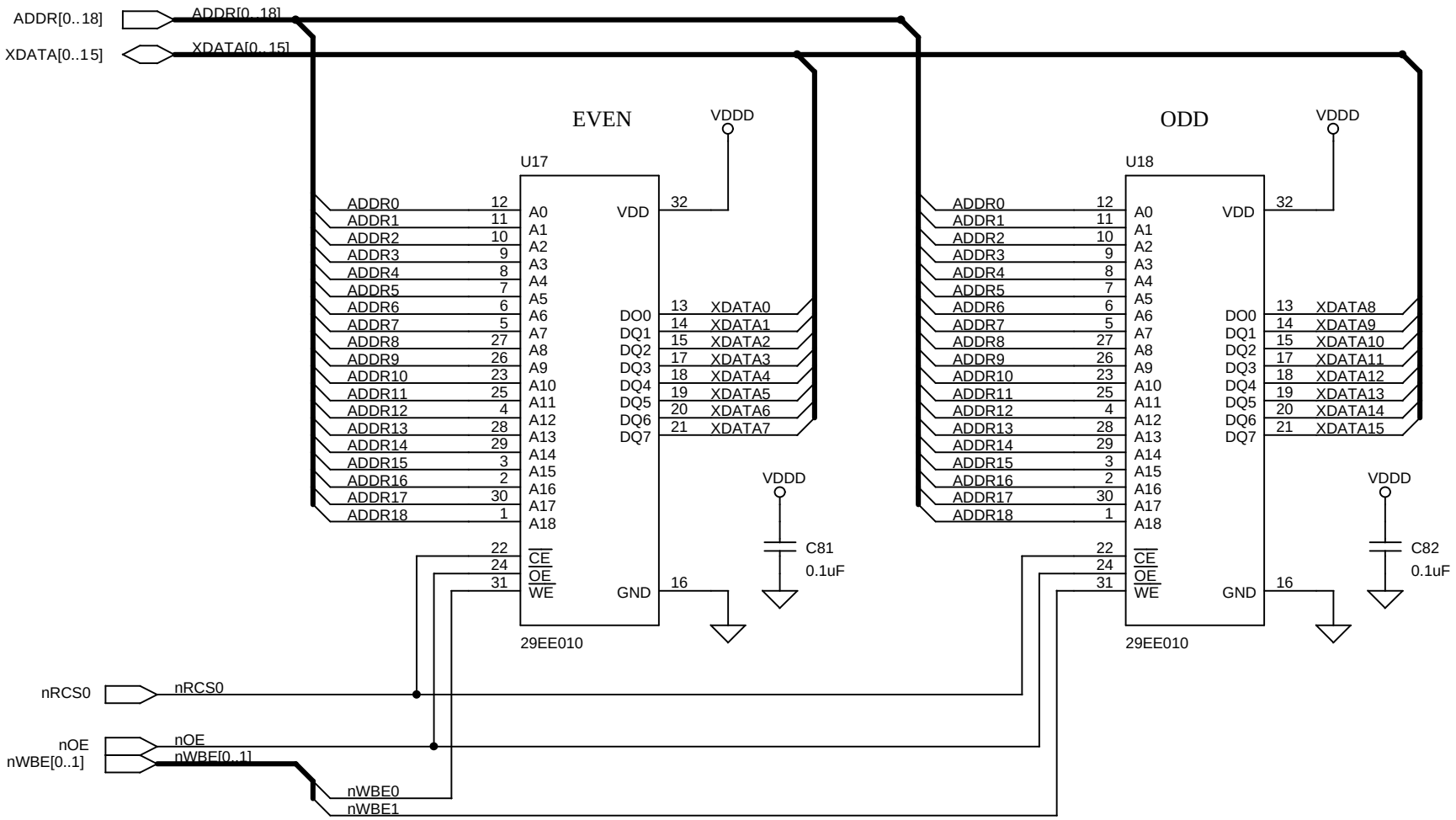


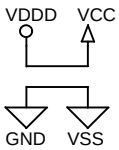
SAMSUNG ELECTRONICS CO.LTD,.		
Title: SNDS100 (Samsung NetARM Development System)		
Size: B	Document Number: EXTERNAL.SCH	Rev: 1.0
Date: Saturday, January 16, 1999	Sheet 4 of 10	


HDLC Serial Communication Port #A, #B
VDDD
R37 680 LED D14 nDTRA
R38 680 LED D15 nDSRA
R39 680 LED D16 nRTSA
R40 680 LED D17 nCTSA
R41 680 LED D18 nDCDA
(DSRA : From P16)
JP5
HEADER 10X2
TXCA RXCA TXDA RXDA nDTRA nDSRA nRTSA nCTSA nDCDA
GND
JP6
HDLCCLK
HEADER 4X2
(DSRB : From P17)
JP7
RXCB TXCB RXDB TXDB nDSRB nDTRB nCTSB nRTSB nDCDB
R42 680 LED D19 nDTRB
R43 680 LED D20 nDSRB
R44 680 LED D21 nRTSB
R45 680 LED D22 nCTSB
R46 680 LED D23 nDCDB
HDLC clock OSC
U14
NC GND GND1 VCC OUT OUT1
OSC
L6 F.B.
C53 0.1uF
R47 22 HDLCCLK
U9
U10
U11
U12
U13
C1+ C1- T1I R1O T2I R2O C2+ C2- VCC V+ T1O R1I T2O R2I V- GND
MAX232
C32 C33 C34 C35 C36 C37 C38 C39 C40 C41 C42 C43 C44 C45 C46 C47 C48 C51 C52 C57 C49 C50 C54 C55 C56 0.1uF
change RXCA, TXDA pin
change nDSRA, nRTSA pin
change RXCB, TXDB pin
change nDSRB, nRTSB pin
RS232C
DTE SIDE(MAIL)
P1A
P1B
CONNECTOR DB25
TXDA RXDA nRTSA RXCA nCTSA nDSRA nDTRA TXCA
TXDB RXDB nRTSB RXCB nCTSB nDSRB nDTRB TXCB
VDDD VCC
GND VSS
SAMSUNG ELECTRONICS CO., LTD.
Title
SNDS100(Samsung NetARM Development System)
Size B
Document Number HDLC.SCH
Rev 1.0
Date: Saturday, January 23, 1999
Sheet 5 of 10

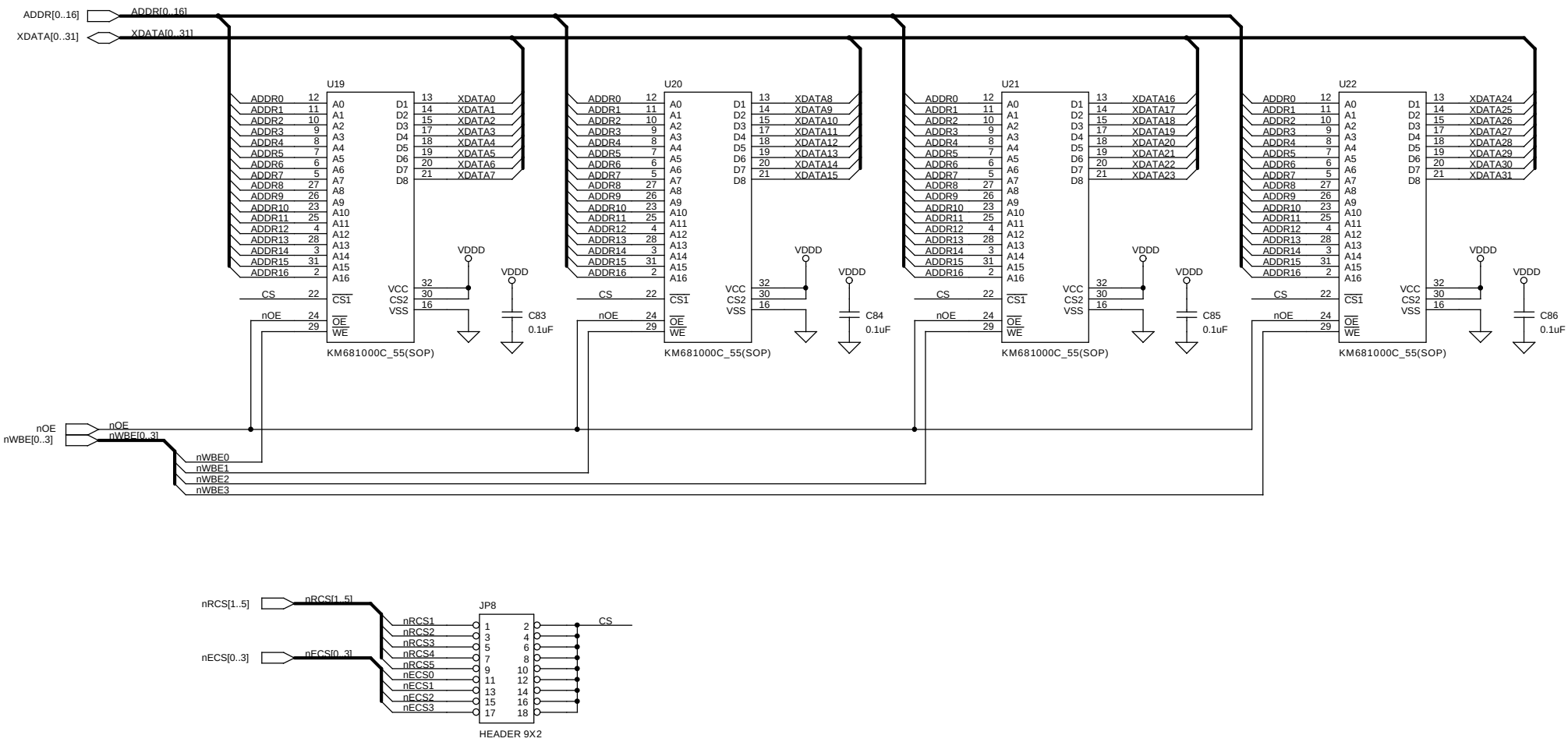

HDLC Serial Communication Port #A, #B
VDDD
R37 680 LED D14 nDTRA
R38 680 LED D15 nDSRA
R39 680 LED D16 nRTSA
R40 680 LED D17 nCTSA
R41 680 LED D18 nDCDA
(DSRA : From P16)
TXCA RXCA TXDA RXDA nDTRA nDSRA nRTSA nCTSA nDCDA
JP5
HEADER 10X2
GND
JP6
HDLCCLK
HEADER 4X2
(DSRB : From P17)
RXCB TXCB RXDB TXDB nDSRB nDTRB nCTSB nRTSB nDCDB
JP7
HEADER 10X2
R42 680 LED D19 nDTRB
R43 680 LED D20 nDSRB
R44 680 LED D21 nRTSB
R45 680 LED D22 nCTSB
R46 680 LED D23 nDCDB
HDLC clock OSC
U14
NC GND GND1 VCC OUT OUT1
OSC
L6 F.B.
C53 0.1uF
R47 22 HDLCCLK
U9
U10
U11
U12
U13
C1+ C1- T1I R1O T2I R2O C2+ C2- VCC V+ T1O R1I T2O R2I V- GND
C32 C33 C34 C35 C36
C37 C38 C39 C40 C41
C42 C43 C44 C45 C46
C47 C48 C51 C52 C57
C49 C50 C54 C55 C56
0.1uF
MAX232
change RXCA, TXDA pin
change nDSRA, nRTSA pin
change RXCB, TXDB pin
change nDSRB, nRTSB pin
RS232C
DTE SIDE(MAIL)
P1A
P1B
CONNECTOR DB25
VDDD VCC
GND VSS
SAMSUNG ELECTRONICS CO., LTD.
Title
SNDS100(Samsung NetARM Development System)
Size B
Document Number HDLC.SCH
Rev 1.1
Date: Friday, May 14, 1999
Sheet 5 of 10

Boot Flash ROM : 8/16 Bit Bus Width is Selectable





SAMSUNG ELECTRONICS CO .LTD,.		
Title: SNDS100 (Samsung NetARM Development System)		
Size: A	Document Number: ROM.SCH	Rev: 1.0
Date: Saturday, January 16, 1999	Sheet 7 of 10	

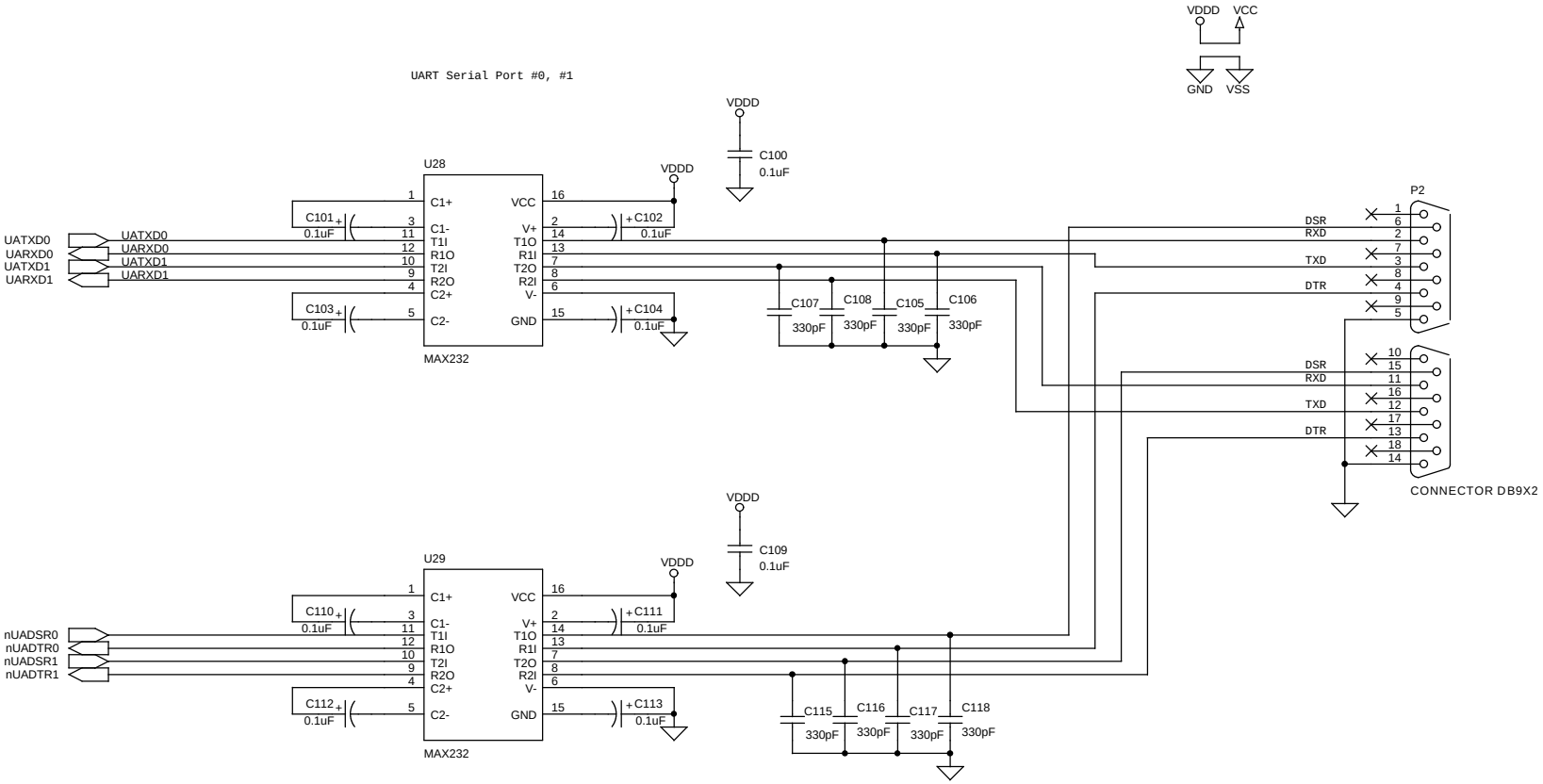


SAMSUNG ELECTRONICS CO.LTD,.

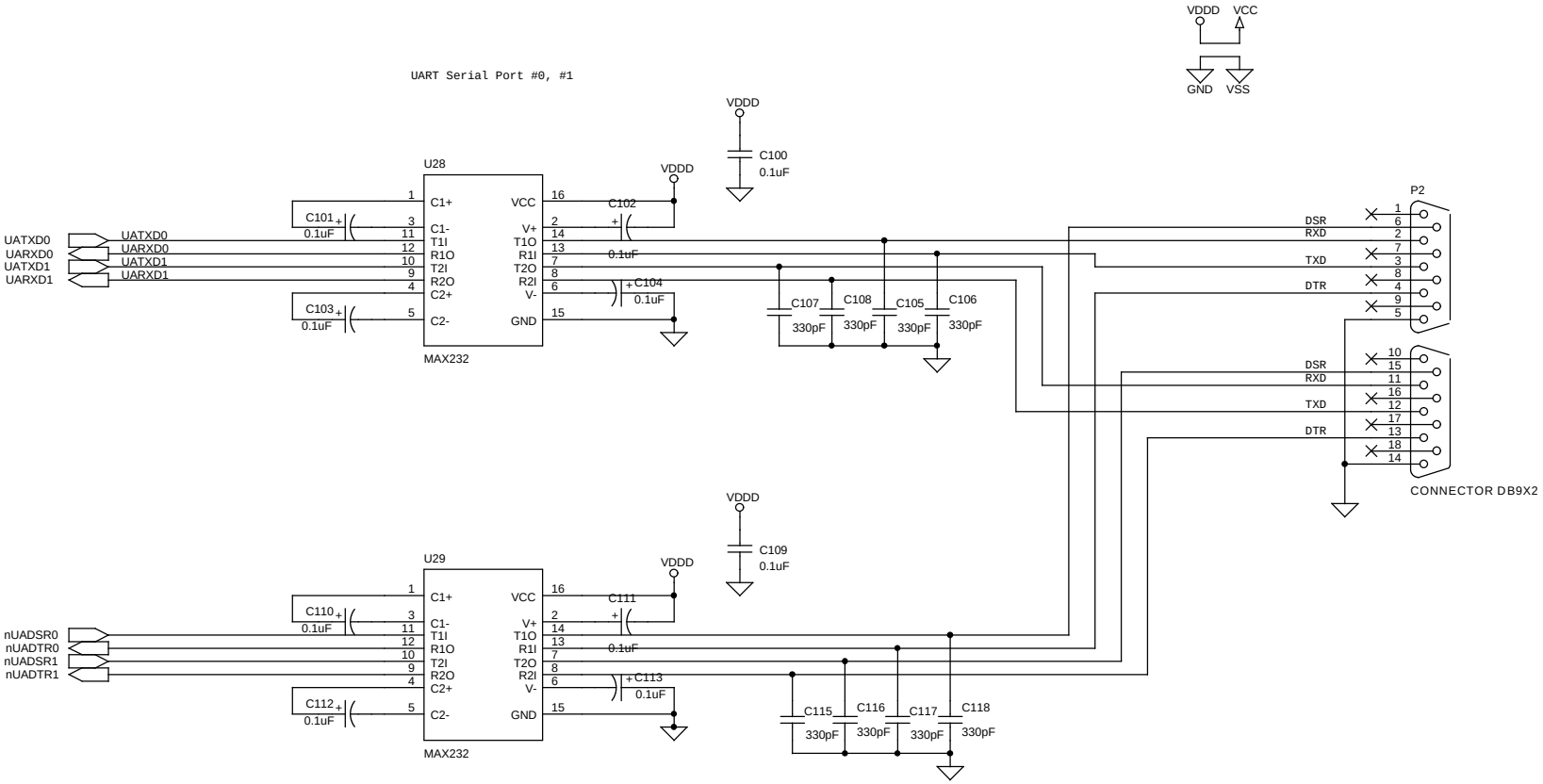
Title: SNDS100 (Samsung NetARM Development System)

Size	Document Number	Rev
B	SRAM.SCH	1.0

Date: Saturday, January 16, 1999 Sheet 8 of 10



SAMSUNG ELECTRONICS CO.LTD,.		
Title: SNDS100 (Samsung NetARM Development System)		
Size: B	Document Number: UART.SCH	Rev: 1.0
Date: Saturday, January 16, 1999	Sheet 10 of 10	



SAMSUNG ELECTRONICS CO.LTD,.		
Title: SNDS100 (Samsung NetARM Development System)		
Size: B	Document Number: UART.SCH	Rev: 1.1
Date: Friday, May 14, 1999	Sheet 10 of 10	

2 HOW TO USE SNDS100 BOARD?

SETUP SNDS100 ENVIRONMENTS

The evaluation environments for SNDS100 are shown in Figure 2-1. Serial port(SIO-0) on SNDS100 have to be connected to COM port of Host PC . This is can be used as console for monitoring and debugging SNDS100.

If you have the emulator like as EmbeddedICE, you can use JTAG port on SNDS100 as the interface for it. Evaluating the 10/100M ethernet function , RJ45 connector on SNDS100 can be connected to Host PC or HUB and Switch. Connected to Host PC, 10/100M ethernet cable has crossed wire connections because of RJ45 connections for SNDS100 were designed to adapter side. In case of HUB/Switch, SNDS100 can be connected to it directly.

DC power adapter which have more then DC6V/800mA output characteristics can be used as input power of the SNDS100 board. This input power regulated to 3.3V and 5.0V for CPU and peripheral device on SNDS100 board.

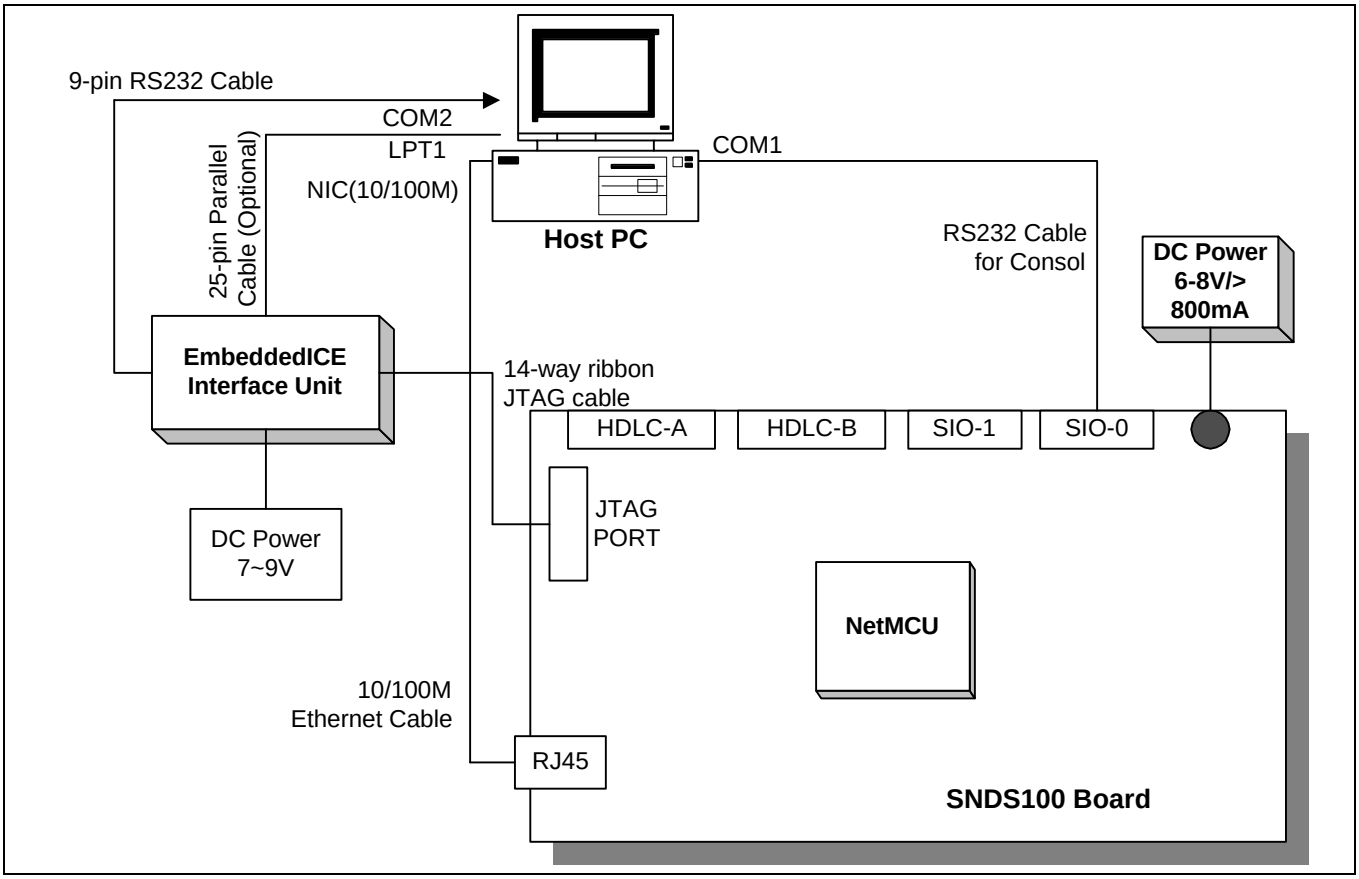
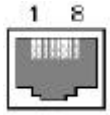


Figure 2-1. Setup Environments for SNDS100 Board

ETHERNET 10/100 BASE-T CONNECTOR

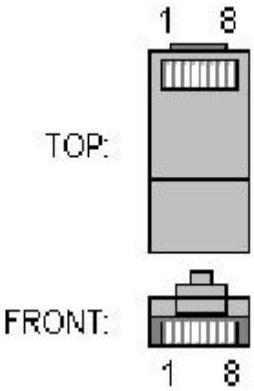
Same connector and pinout for both 10Base-T and 100Base-Tx.



(At the network interface cards/hubs)

(At the cables)

RJ45 FEMALE CONNECTOR at the network interface cards and hubs .
RJ45 MALE CONNECTOR at the cables.



Pin	Name	Descriptions
1	TX+	Transmit Data+
2	TX-	Transmit Data-
3	RX+	Receive Data+
4	N/C	Not Connected
5	N/C	Not Connected
6	RX-	Receive Data-
7	N/C	Not Connected
8	N/C	Not Connected

NOTE: TX & RX are swapped on Hub

CONNECTION METHODS FOR UTP CABLE

RJ45 pins on SNDS100 is defined to Adapter side. So, you can straight connect SNDS100 to HUB through UTP cable. In this case, between the SNDS100 and Hub, the pin numbers are correspond to each others.

Between the SNDS100 board and NIC which is on Host PC, you have to connect each other through UTP cable which is crossover patch cord. See Figure 2-2 (b)

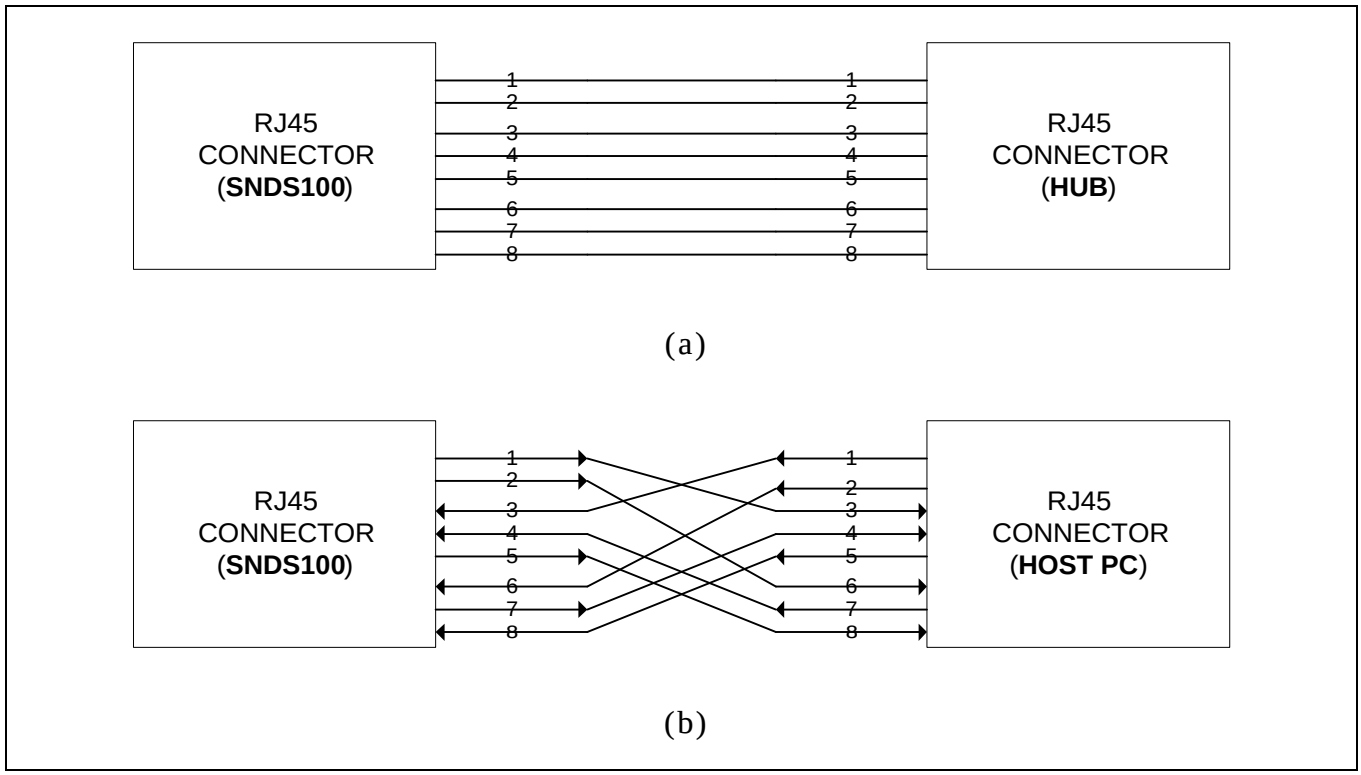


Figure 2-2. UTP Cable Connections

CONNECTION CONFIGURATIONS FOR DEBUG CONSOL

UART channel 0(SIO-0) on SNDS100 is assigned to Debug Console port. Using this port, you can download executable image code to DRAM memory from DOS command windows of Host PC . Also you can monitoring and debugging the device from this port with Hyper terminal which is windows utility program supported by windows 95 and windows NT.

You can communicate through the RS232 cable from UART channel 0 to COM1 or COM2 which is serial port on Host PC. SNDS100 supplies the 9pin D-SUB male connector for communication channel. Detail pin configurations for connecting each others as console port be given as bellows:

Table 2-1. Pin connection configurations for Console port.

SNDS100	DIR	HOST PC		DESCRIPTIONS
9Pin D-SUB MALE		9Pin D-SUB MALE	25Pin D-SUB MALE	
2. RXD	←	2. RXD	3. RXD	Receive Data
3. TXD	→	3. TXD	2. TXD	Transmit Data
4. DTR	→	4. DTR	20. DTR	Data Terminal Ready
5. DSR	←	5. DSR	6. DSR	Data Set Ready

CONFIGURING THE HYPER TERMINAL (ON WINDOWS 95 OR NT)

To configure the Hyper Terminal which is windows utility program for serial communications be given on windows 95 or NT, please following steps:

1. Running the Hyper Terminal utility program.

   Window 95 or windows NT start tool bar → Program → Accessories → Hyper Terminal Group → Double click Hyperterm.exe → Enter connection name → Select icon → Click OK.

2. Select COM port to communicate with SNDS100 target board.

   Choose COM1 or COM2 as the serial communication port → Click OK

3. Set the serial port properties. (See the Figure 2-3.)

— Bits per second : 115200 bps
— Data bits : 8 bits
— Stop bits : 1
— Flow control : None

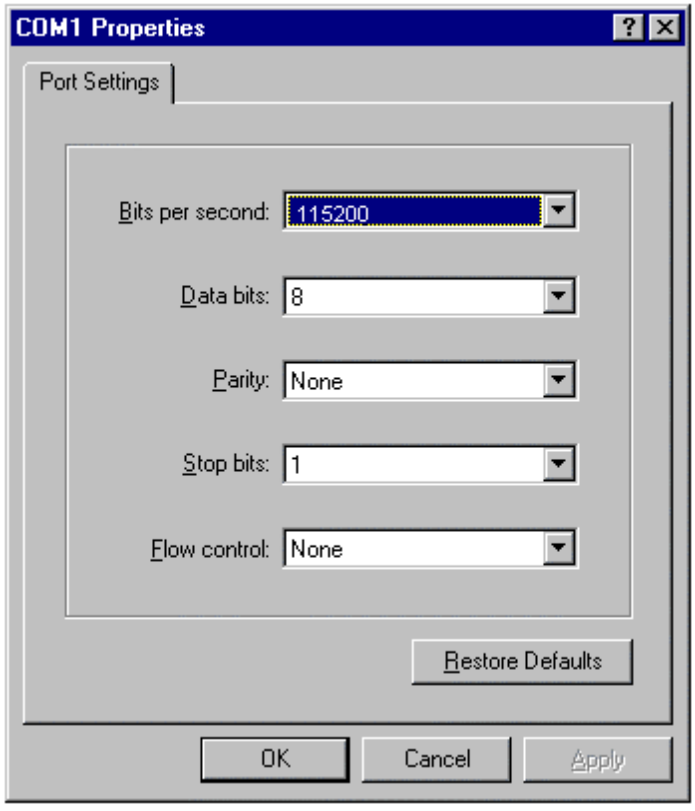


Figure 2-3. Setting Port Properties

CONFIGURING THE HYPER TERMINAL (ON WINDOWS 95 OR NT) (Continued)

4. Select Properties menu

   File → Properties.

5. Choose Setting Page (Figure 2-4)

6. Click ASCII Setup button

7. Check ASCII Setup menus and set (Figure 2-5)

8. Re-connect Hyper-Terminal to run at new properties

— Disconnect: Call → Disconnect
— Connect : Call → Call

9. Power-On Reset or Push the reset button on SNDS100 board

— Now, The diagnostic menu is showed on the Hyper-Terminal (Refer to Figure 2-6).

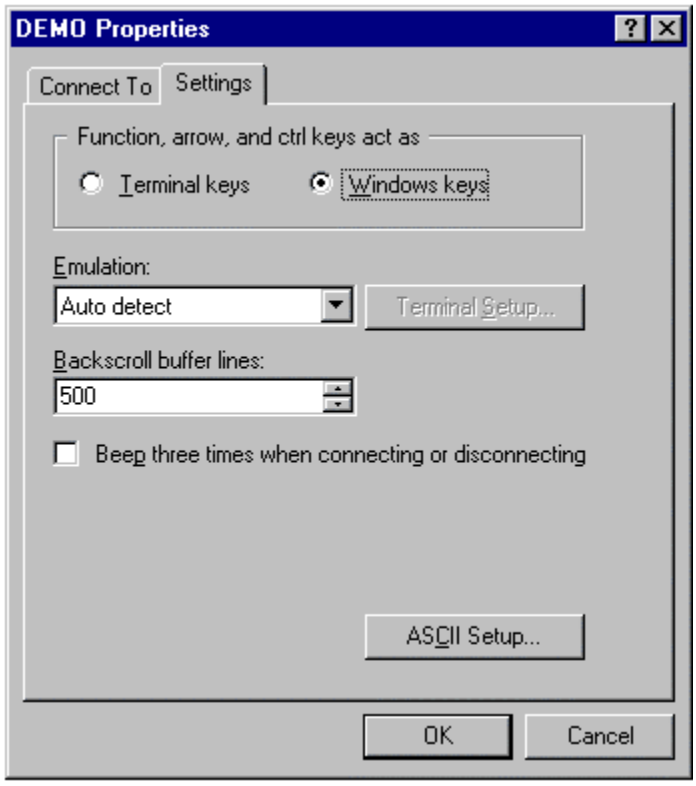


Figure 2-4. Choose Setting Page

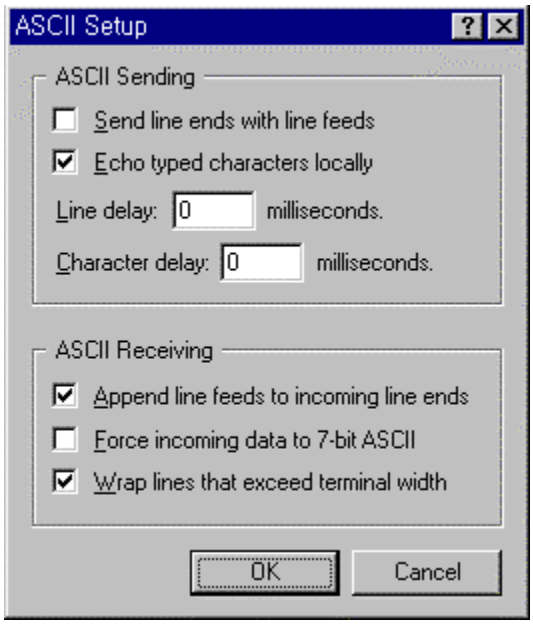


Figure 2-5. ASCII Setup Menu

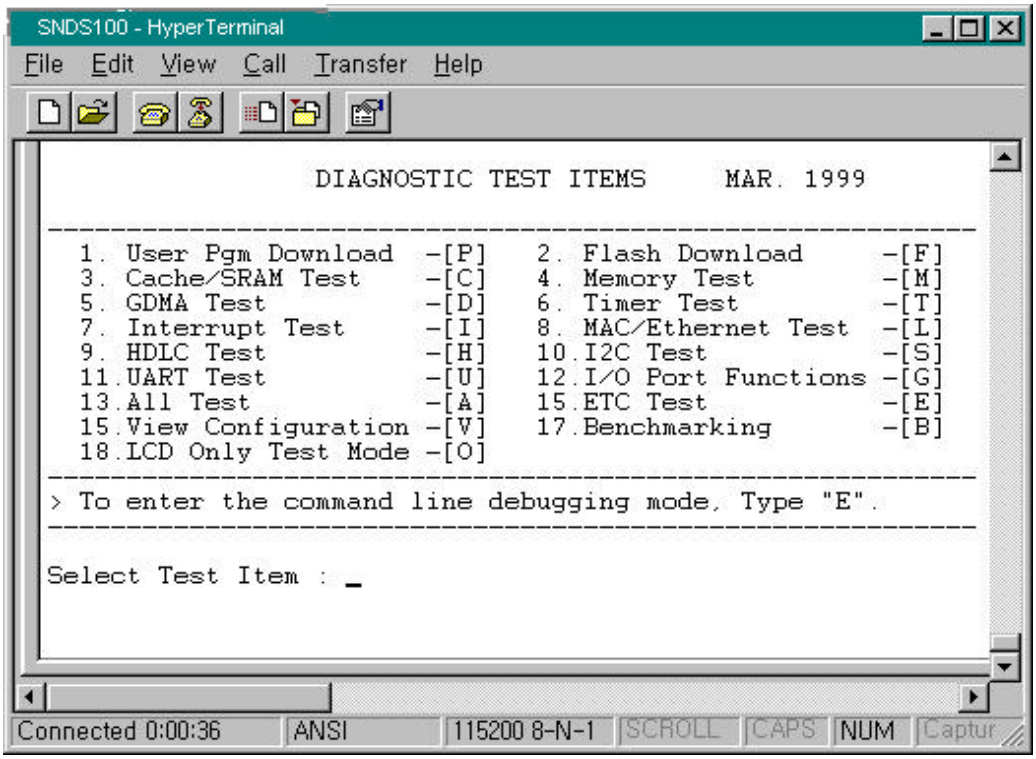


Figure 2-6. SNDS100 Diagnostic Window After Reset

HOW TO BUILD EXECUTABLE DOWNLOAD IMAGE FILE

Execution image file can be built using by ARM Project manager or makefile. If you want to use ARM debugger, you have to build ARM image format(AIF). Any other case, if you want to use only Console debugging SNDS100 board without ARM debugger, then Binary image file have to be prepared to download directly and execute on Target board.

First of all, you must download SNDS100.zip which is evaluation source included boot code from our web site(www.samsungsemi.com) and also any other utilities and follow up the bellows. It will be helpful to more easily understand the development environments of NetMCU series.

USING ARM PROJECT MANAGER

ARM SDT version 2.11a were installed and used for the following procedures.

Creating a New Project

To create a new project, following the steps:

1. Invoke ARM Project Manager

2. Generate a New Project

   File menu → New → select Project form the New dialog box

3. Edit New Project dialog box (Figure 2-7)

— Type : ARM Executable image
— Enter Project Name and Directory

4. When you have created a new project, the information in the Project Window is displayed in a hierarchical flow diagram. (Figure 2-8)

— Debug: for creating a target image suitable for debugging, which includes debugging information.
— Release: for creating a target image suitable for release, which includes debugging information.

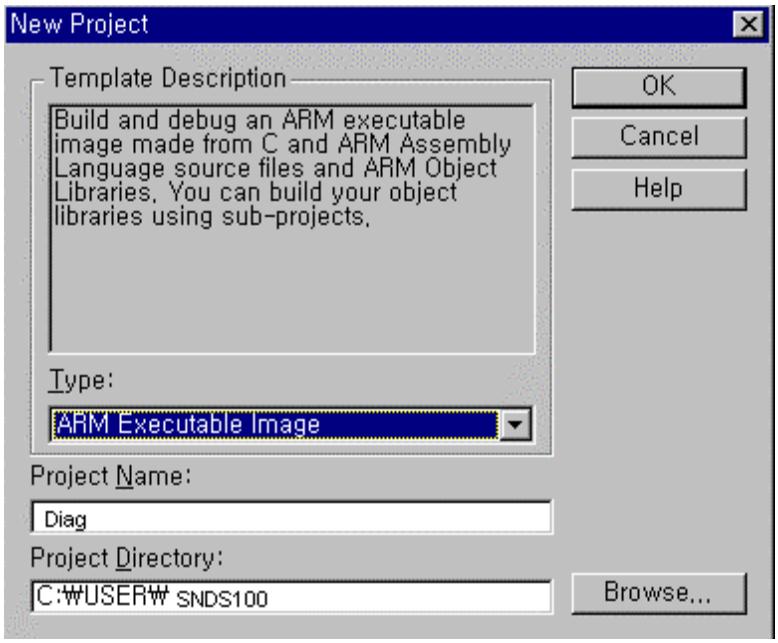


Figure 2-7. New Project Dialog Window

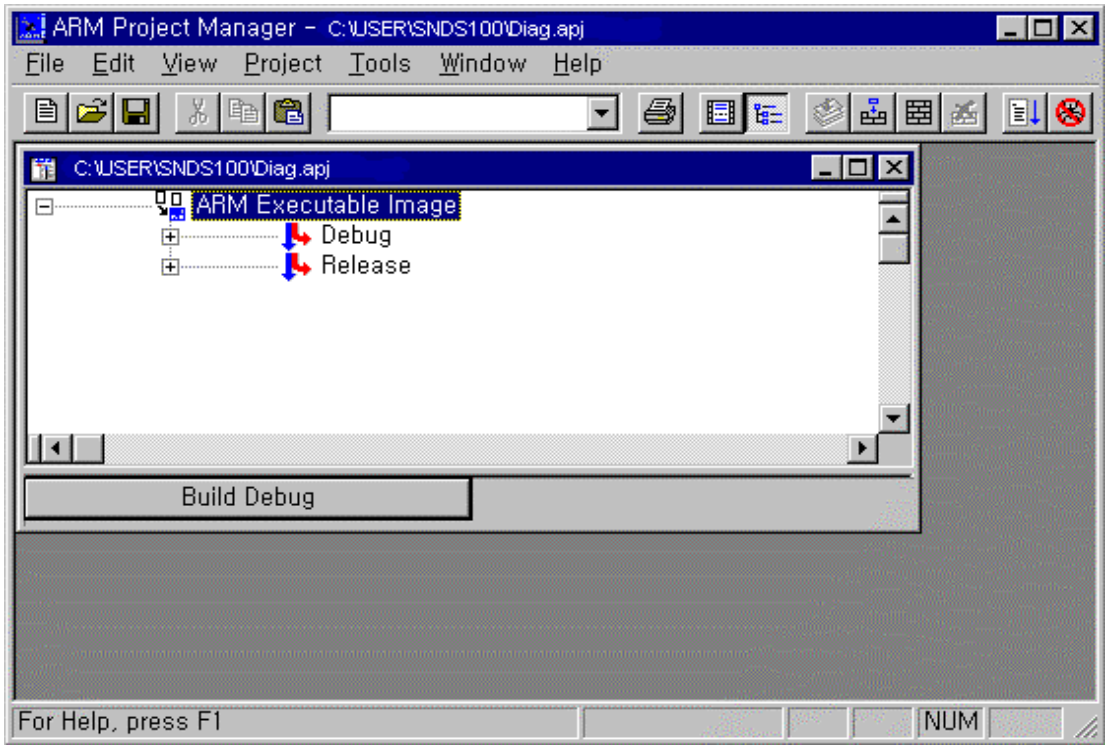


Figure 2-8. Project Window

ADDING SOURCE FILES TO THE PROJECT

To add source files to the project, following the steps:

1. Select Add files to Project from the Project menu.
2. Select source files to add (*.c and *.s on working directory)

SETTING THE OPTIONS OF ARM PROJECT MANAGER

You should set the options on Tools menu for setting ARM Project Manager tool.For additional options and descriptions of compiler, assembler, and linker, please refer to the chapter1,2, and 3 of “ARM Software Development Toolkit Reference Guide”.

Setting C Compiler option

1. Select <cc>=armcc option.

   Tools -> Configure -> <cc>=armcc

2. If the modify warning dialog box pops up, read it, and click Yes.

3. Modify Target page on Compiler Configuration dialog box as shown in Figure 2-9.

— Addressing Mode: 32 Bit
— Processor: ARM7TM
— Byte Sex: Big Endian

4. Write option to C&Debug page on Compiler Configuration dialog box as shown in Figure 2-10.

— Extra command line arguments: -fc

5. Click OK.

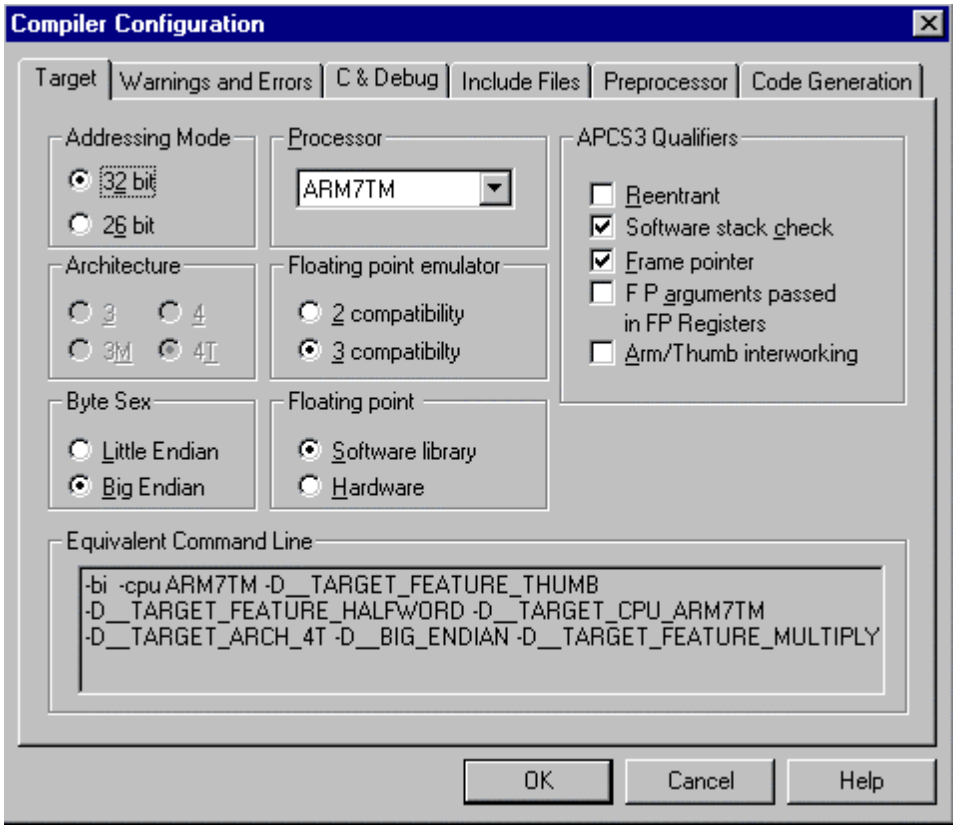


Figure 2-9. Compiler Configuration: Target Page

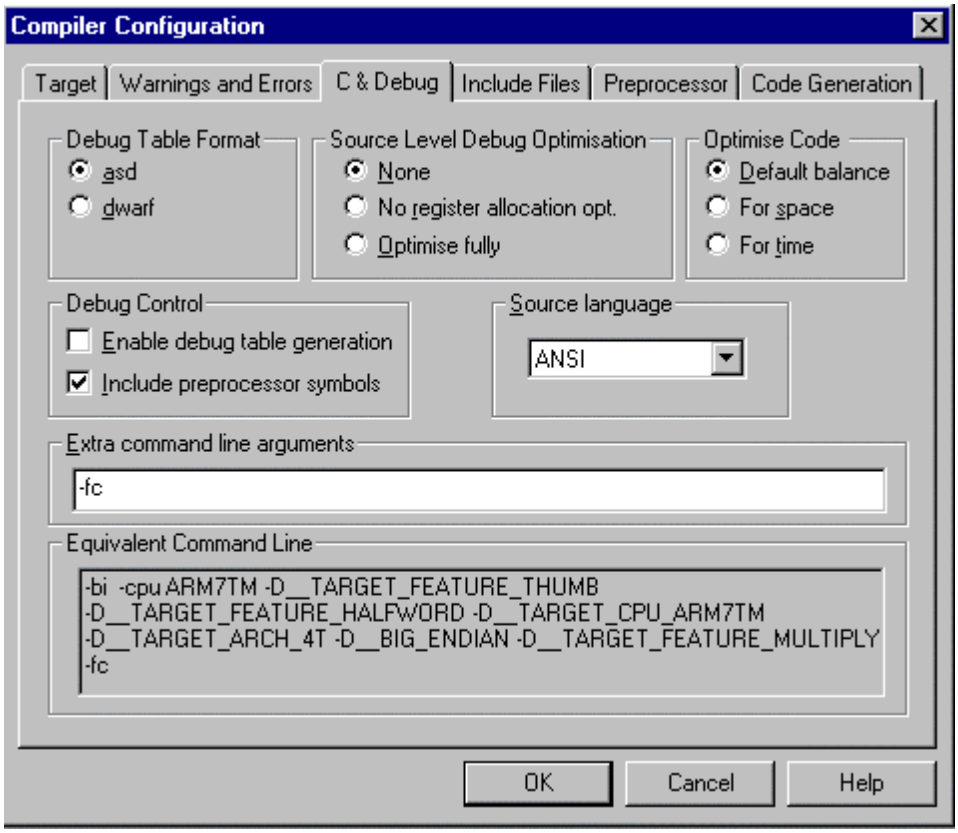


Figure 2-10. Compiler Configuration: C & Debug page

Setting Assembler Option

1. Select <asm>=armasm option.

   Tools -> Configure -> <asm>=armasm

2. If the modify warnning dialog box pops up, read it, and click Yes.

3. Modify Target page on Assembler Configuration dialog box (Figure 2-10).

— Processor: ARM7TM
— Byte Sex: Big Endian

4. Click OK.

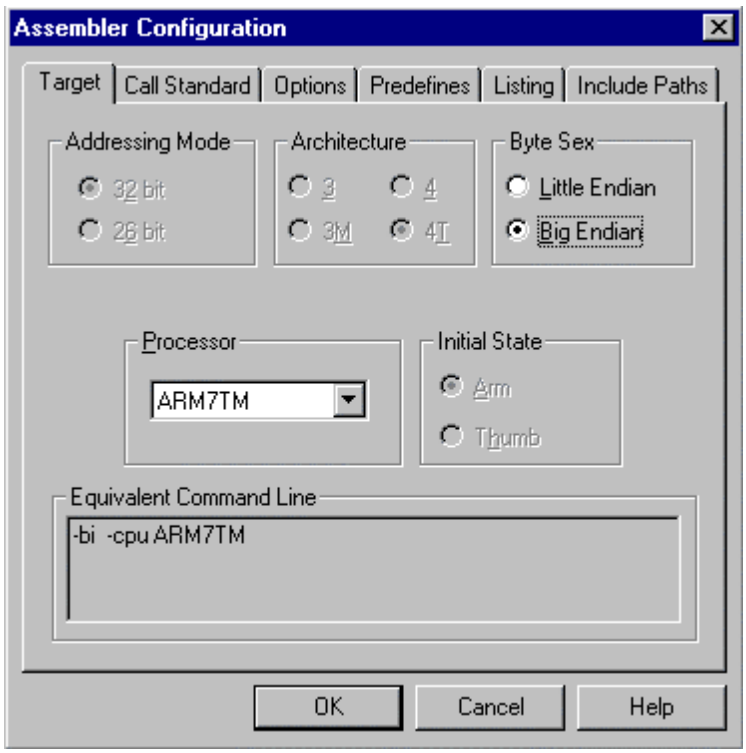


Figure 2-11. Assembler Configuration

Setting Link Option

1. Select armlink option.

Tools -> Configure -> armlink

2. If the modify warning dialog box pops up, read it, and click Yes.

3. Modify Output page on Linker Configuration dialog box (Figure 2-12).
   Output Formats: Absolute AIF

4. Modify Entry and Base page on Linker Configuration dialog box (Figure 2-13).

— Read-Only: 0x1000050

— Read-Write: 0x1300000

5. Modify ImageLayout page on Linker Configuration dialog box (Figure 2-14).

— Object File: init.o

— Area Name: init

4. Click OK.

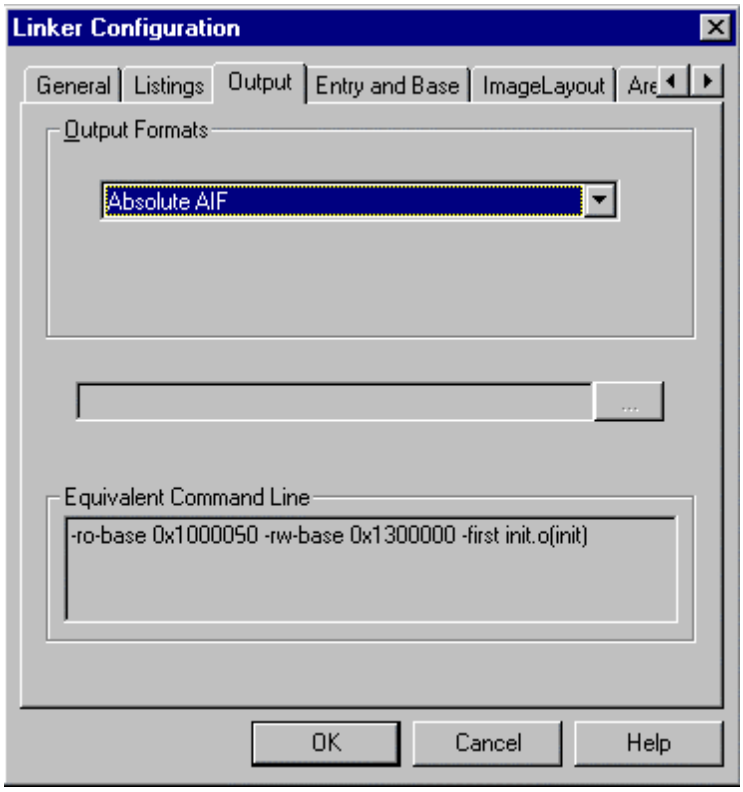


Figure 2-12. Linker Configuration: Output Page

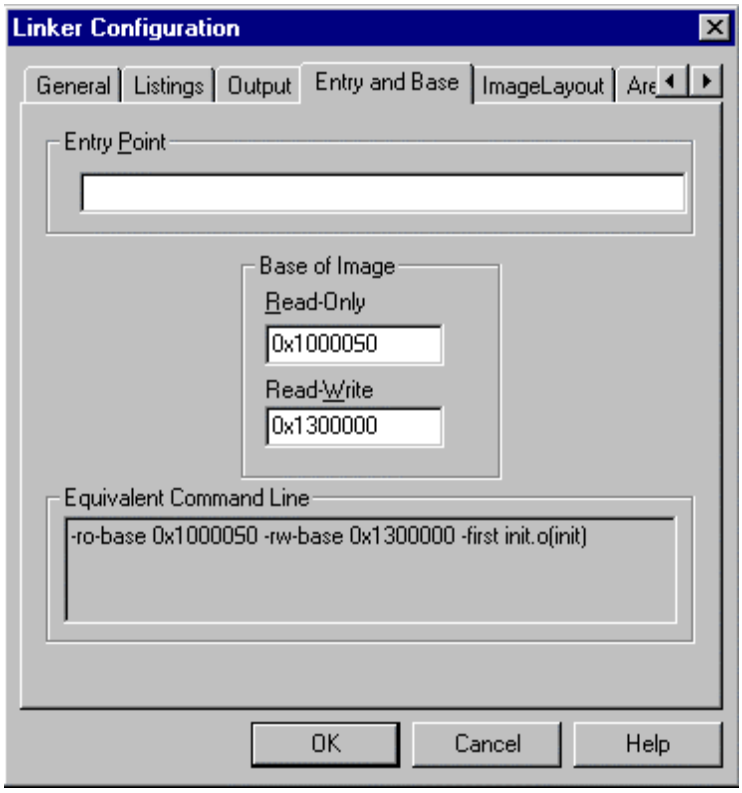


Figure 2-13. Linker Configuration: Entry and Base Page

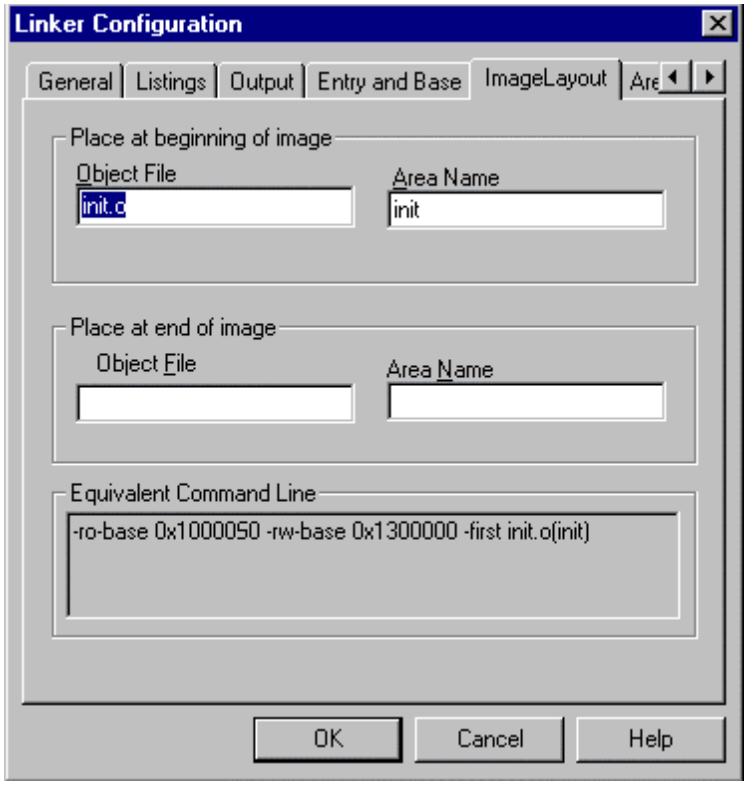


Figure 2-14. Linker Configuration: Image Layout Page

BUILDING DIAG PROJECT

Now, You can start compiling the diag source code in ARM Project manager and Check error message or warning message, and fix problem. If there is no compiling error, executable image file named diag.axf or diag.bin can be generated at working directory.

If the generated image format is ARM image format(ex. Diag.axf), you can execute it by click ARM debug icon on windows or typing "adw diag.axf " at DOS windows. Adw is ARM debugger execution file.

If the generated image format is Binary format, then you can use sftp.exe utility which is available at web site in DOS windows. It`s usage is described at the end of this section.

HOW TO USING ARM DEBUGGER WITH EMBEDDED ICE

If you have built a Diag project without any error, find the executable ARM image output file (diag.axf) on your project sub-directory. Then, execute and debug the project. The generated image file will be downloaded by ARM debugger through the EmbeddedICE(ARM Emulator) interfacing JTAG port to DRAM memory on the SNDS100 target board. Next, you can start to debug the downloaded image using ARM Debugger Window(ADW).

STARTING ARM DEBUGGER

1. Reset the SNDS board and the EmbeddedICE interface unit.

   Reset SNDS (Press reset switch) -> Reset EmbeddedICE interface unit (Press red switch)

2. Start ARM Debugger tool from ARM Project Manager window.

   Project menu -> Debug Diag.apj

   Also, you can start it at DOS windows as you type "adw diag.axf " in there.

3. When ARM Debugger is started, it will load the image code to ARMulator (refer to section 1.5).

   If you ever used ARM Debugger as a remote debugger, Remote Debugger warning message dialog box will be displayed. If the remote debugger option is correct, then select Yes, otherwise, select No.

CONFIGURING ARM DEBUGGER

In order to access a remote target, you should configure ARM Debugger for Windows (ADW) rather than ARMulator. The EmbeddedICE interface unit must also be configured for the ARM core in the target system (SNDS100 board).

The ARM7TDMI core is contained in the KS32C5000 /5000A or KS32C50100 on the SNDS100 board. ARM7TDMI macro cell includes the ARM7 core with Thumb, debug extensions, and the EmbeddedICE macro cell.

To configure ARM Debugger using the EmbeddedICE interface, following the steps:

1. Select Configure Debugger from the Options menu.

Options -> Configure Debugger

2. Debugger Configuration dialog box is displayed (Figure 2-15). When you select Target page, there are two Target Environment available as follows.

— ARMulator: lets you execute the ARM program without any physical ARM hardware by simulating ARM instructions in software.

— Remote_A: connects the ARM debugger directly to the target board or to an EmbeddedICE unit attached to the target.

3. Select Remote_A from target environments, and click Configure.

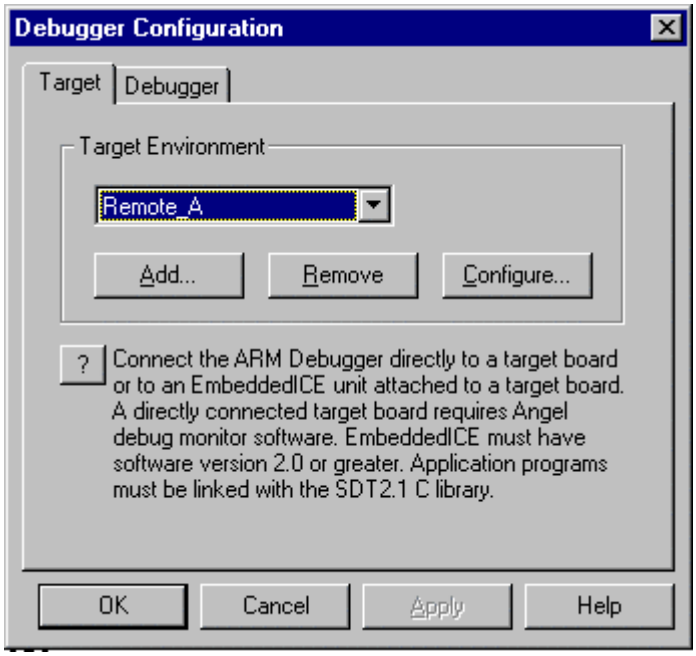


Figure 2-15. Debugger Configuration: Target Page

CONFIGURING ARM DEBUGGER (Continued)

4. Configure Angel Remote Configuration dialog box(Figure 2-16).

— Remote Connection: select your host and EmbeddedICE communication port configuration.

— Select Ports and Serial Line Speed.

5. Select Debugger page from Debugger Configuration dialog box (Figure 2-17) and configure it.

— Endian: big

— Default Memory Map: erase the contents of the default memory map.

6. If you click the OK button on Debugger Configuration dialog box, the debugger will be restarted. The restarting dialog box is displayed and numbers are rapidly changing, indicating that it is reading and writing to target. This means that the executable image file is downloaded to the DRAM code area.

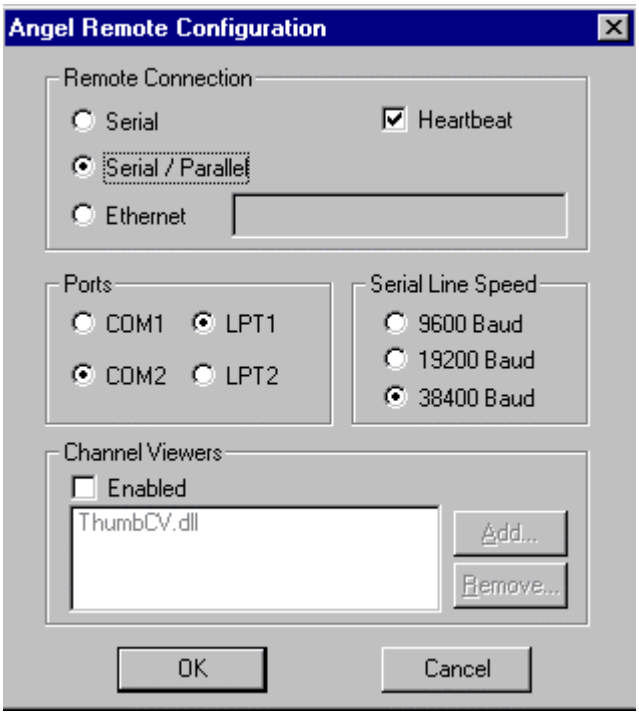


Figure 2-16. Angel Remote Configuration

CONFIGURING ARM DEBUGGER (Continued)

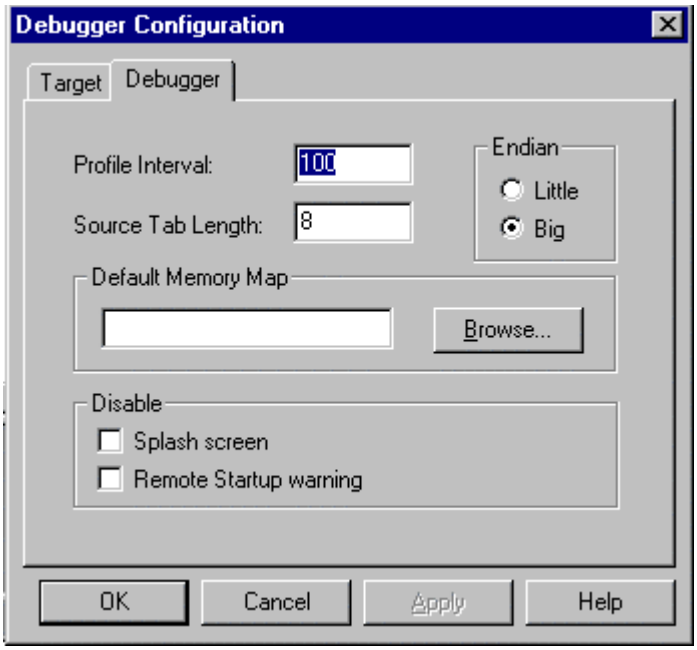


Figure 2-17. Debugger Configuration: Debugger Page

7. When download is finished, select Configure EmbeddedICE from the Options menu.

   Options -> Configure EmbeddedICE

8. EmbeddedICE Configuration dialog box is displayed (Figure 2-18).

— Name: ARM7TDI

9. If you select the OK button in EmbeddedICE Configuration dialog, a new dialog box will be displayed with version number.

10. Click OK.

CONFIGURING ARM DEBUGGER (Continued)

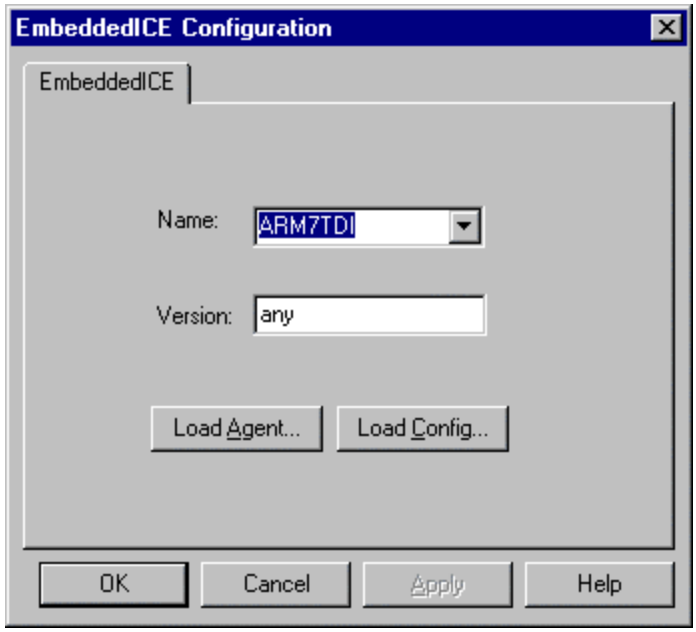


Figure 2-18. EmbeddedICE Configuration

EXECUTING DIAG PROJECT IMAGE

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

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

Or, you can initialize these variables as follows:

First, create a text file named "armsd.ini", which includes the commands described above. Then, enter the following command in the Command window (Figure 2-19):

```
obey c:\arm211\armsd.ini
```

For more information about these steps, please refer to Chapter 6 of "ARM Software Development Toolkit User's Guide".

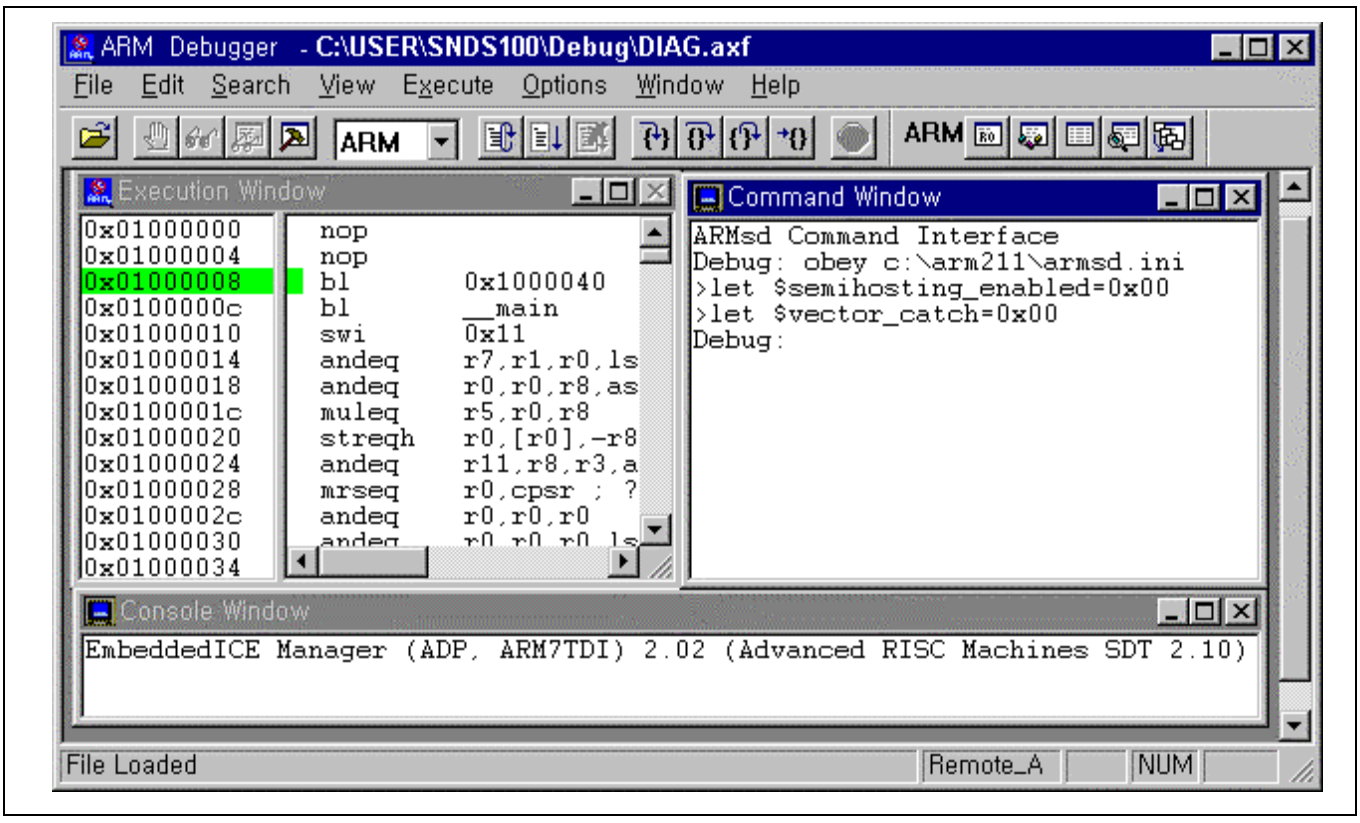


Figure 2-19. ARM Debugger Window(ADW): Command Window

2. Execute the program.

   Execute menu -> Go

3. Now, The downloaded image file(diag.axf) will be run on DRAM area. Diagnostic program running status can be monitored on the current Hyper-Terminal diagnostic window.

DEBUGGING DOWNLOAD IMAGE IN ADW

STEPPING THROUGH THE PROGRAM

To step through the program execution flow, you can select one of 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.

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

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

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

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 View menu.
2. Double-click the function name you want to open. A new source window is displayed containing the function source.
3. Double-click 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. (The program stops only when this expression is correct).

SETTING A WATCH POINT

A watch point halts a program when the value of a specified register or a variable changes or is set to a specific number.

To set a watch point, follow these steps :

1. Display a list of registers, variables, and memory locations you want 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 want to set the watchpoint. Then choose Set or Edit Watchpoint from the Execute menu.
3. Enter a Target Value in the Set or Edit Watchpoint dialog box. Program operation will stop 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 related heading from the View menu:

— 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

If you want to display the source code interleaved with disassembly, choose Toggle Interleaving on the Options menu. This command toggles between Displaying Source Only and Displaying Source Interleaved With Disassembly. When the source code is shown interleaved with disassembly, machine instructions appear in a lighter gray color.

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

USING MAKEFILE FOR IMAGE BUILD

Now, you know how to build image debug and execute use by ARM SDT on windows environments. And also, you can build the download image file with makefile on DOS environments. Using this makefile, you can generate not only ARM image format but Binary image.

If you have no emulator like as EmbeddedICE, you can simply debug and evaluate the target board with this binary image. At DOS window, this binary image file can be download to target board through serial cable using sftp.exe utility which is available at our web site. More detail refer to the later in this section.

Sometimes, it is more convenience for engineer to build image at DOS windows not using GUI of ARM SDT. The sample makefile for diagnostic sources including boot code is shown bellows:

Makefile

```
#########################################################################
#                                                                       #
#       SNDS100 EVALUATION BOARD ENVIRONMENTS SETUP                      #
#         (KS32C5000/KS32C5000A/KS32C50100)                             #
#                                                                       #
#########################################################################
#                                                                       #
#  !! EDIT THE BELLOW THE OPTION FLAG FOR TO FIT YOUR SYSTEM.           #
#       ( PLEASE,READ "README.TXT" FILES FOR GUIDE )                    #
#                                                                       #
#########################################################################
# ------------------------------------------------------------------ #
#                 ARM TOOLKIT ENVIRONMENTS
# ------------------------------------------------------------------ #
ARMPATH   = C:\ARM211A
ENDIAN    = -BI
ARM_LIB   = ARMLIB_CN.32B
C         = ARMCC
ASSEMBLER = ARMASM
INTERWORK =
IMAGE     = AIF
CPU_OPTS  = -PROCESSOR ARM7TM -ARCH 4T
DEVICE    = KS32C50100

#-----------------------------------------------------------------#
#             ASSEMBLER PREDEFINED FLAGS                          #
#-----------------------------------------------------------------#
#IF(ROM_ADDRESS_AT_ZERO)
#{
#APDFLAG   = TRUE
#MEMORY    = -RO 0X0 -RW 0X1300000
#MEMORY    = -RO 0X0
#}
#ELSE
#{
APDFLAG   = FALSE
MEMORY    = -RO 0X1000050 -RW 0X1300000
#}
```

```
#-----------------------------------------------------------------#
#             C-PREDEFINED OPTION FLAGS                           #
#-----------------------------------------------------------------#
CPDFLAGS     = -D$(DEVICE)  -DEX_UCLK
#CPDFLAGS     = -D$(DEVICE)


###########################################################################
#                                                                         #
# !!!DON'T TOUCH THIS MAKEFILE, EXCEPT FOR YOU HAVE TO CHANGE IT.!!  #
#                                                                         #
###########################################################################
#                                                                         #
#       SNDS100 EVALUATION BOARD'S MAKEFILE FOR IMAGE BUILD             #
#         (KS32C5000/KS32C5000A/KS32C50100)                               #
#                                                                         #
###########################################################################

LINK         = $(ARMPATH)\BIN\ARMLINK
ASM          = $(ARMPATH)\BIN\$(ASSEMBLER)
CC           = $(ARMPATH)\BIN\$(C)
ASM4ARM      = $(ARMPATH)\BIN\ARMASM
CC4ARM               = $(ARMPATH)\BIN\ARMCC
ASM4THUMB    = $(ARMPATH)\BIN\TASM
CC4THUMB     = $(ARMPATH)\BIN\TCC
LIB          = $(ARMPATH)\LIB\$(ARM_LIB)
INC          = $(ARMPATH)\INCLUDE

ASMFLAGS     = $(ENDIAN) -G -APCS 3/32BIT
ASMFLAG_BOOT = $(ENDIAN) -G -APCS 3/32BIT -PD "ROM_AT_ADDRESS_ZERO SETL {$(APDFLAG)}"
CFLAGS       = $(ENDIAN) -G -C -FC -APCS 3/32BIT $(CPDFLAGS) $(CPU_OPTS) -I$(INC)

#*********************************************************************#
#       LINK  OPTION FLAGS FOR BUILD ROM/DRAM/ICE IMAGE               #
#*********************************************************************#
LINK32_5000 = -FIRST INIT.O(INIT) $(MEMORY) -O DIAG -$(IMAGE) -BIN -NOZEROPAD -SYMBOLS
DIAG.SYM $(BSPLIB1) $(BSPLIB2) $(HDLC) $(ARM) $(LIB) $(BSPTEST)

LINK16_5000 = -FIRST TINIT.O(INIT) $(MEMORY) -O TDIAG -$(IMAGE) -DEBUG -NOZEROPAD -SYMBOLS
DIAG.SYM $(BSPLIB1) $(BSPLIB2) $(HDLC) $(THUMB) $(LIB) $(BSPTEST)

LINK32_50100 = -FIRST INIT.O(INIT) $(MEMORY) -O DIAG100 -$(IMAGE) -DEBUG -NOZEROPAD -SYMBOLS
DIAG.SYM $(BSPLIB1) $(BSPLIB2) $(HDLC100) $(ARM) $(LIB) $(BSPTEST)

LINK16_50100 = -FIRST TINIT.O(INIT) $(MEMORY) -O TDIAG100 -$(IMAGE) -DEBUG -NOZEROPAD -
SYMBOLS DIAG.SYM $(BSPLIB1) $(BSPLIB2) $(HDLC100) $(THUMB) $(LIB) $(BSPTEST)

#*********************************************************************#
#           OBJECT CODES FOR BUILD IMAGE                              #
#*********************************************************************#
THUMB          = TINIT.O TSTART.O
ARM           = INIT.O START.O
HDLC          = HDLC.O HDLCINIT.O HDLCLIB.O
```

```
HDLC100       = HDLCMAIN.O HDLC100INIT.O HDLC100LIB.O
BSPLIB1       = DIAG.O DOWN.O FLASH.O MEMORY.O POLLIO.O UART.O ISR.O TIMER.O
BSPLIB2      = IIC.O SYSTEM.O DMA.O KSLIB.O MAC.O MACINIT.O MACLIB.O IOP.O
BSPTEST       = UART_TEST.O IIC_TEST.O TIMER_TEST.O DHRY_1.O DHRY_2.O LCD.O


DIAG: $(BSPLIB1) $(BSPLIB2) $(HDLC) $(ARM) $(BSPTEST)
    $(LINK) $(LINK32_5000)

TDIAG: $(BSPLIB1) $(BSPLIB2) $(HDLC) $(THUMB) $(BSPTEST)
    $(LINK) $(LINK16_5000)

DIAG100: $(BSPLIB1) $(BSPLIB2) $(HDLC100) $(ARM) $(BSPTEST)
    $(LINK) $(LINK32_50100)

TDIAG100: $(BSPLIB1) $(BSPLIB2) $(HDLC100) $(THUMB) $(BSPTEST)
    $(LINK) $(LINK16_50100)

#*********************************************************************#
#                BUILD OPTIONS FOR BOOT CODE                  #
#*********************************************************************#
INIT.O: INIT.S
        $(ASM4ARM) $(ASMFLAG_BOOT) INIT.S -O INIT.O -LIST INIT.LST
START.O: START.S
        $(ASM4ARM) $(ASMFLAGS)  START.S -O START.O
TINIT.O: TINIT.S
        $(ASM4THUBM) $(ASMFLAG_BOOT) TINIT.S -O TINIT.O -LIST TINIT.LST
TSTART.O: TSTART.S
        $(ASM4THUBM) $(ASMFLAGS)  TSTART.S -O TSTART.O

#*********************************************************************#
#               DIAGNOSTIC SOURCE CODES                       #
#*********************************************************************#
DIAG.O: DIAG.C
        $(CC) $(CFLAGS) -ERRORS DIAG.ERR  DIAG.C
DOWN.O: DOWN.C
        $(CC) $(CFLAGS) -ERRORS DOWN.ERR  DOWN.C
FLASH.O: FLASH.C
        $(CC) $(CFLAGS)  -ERRORS FLASH.ERR FLASH.C
MEMORY.O: MEMORY.C
        $(CC) $(CFLAGS)  -ERRORS  MEMORY.ERR MEMORY.C
POLLIO.O: POLLIO.C
        $(CC) $(CFLAGS)  -ERRORS POLLIO.ERR  POLLIO.C
UART.O: UART.C
        $(CC) $(CFLAGS) -ERRORS UART.ERR  UART.C
UART_TEST.O: UART_TEST.C
        $(CC) $(CFLAGS) -ERRORS UART_TEST.ERR  UART_TEST.C
LCD.O: LCD.C
        $(CC) $(CFLAGS) -ERRORS LCD.ERR  LCD.C
ISR.O: ISR.C
        $(CC) $(CFLAGS)  -ERRORS ISR.ERR ISR.C
TIMER.O: TIMER.C
        $(CC) $(CFLAGS)  -ERRORS TIMER.ERR TIMER.C
```

```
TIMER_TEST.O: TIMER_TEST.C
        $(CC) $(CFLAGS)  -ERRORS TIMER_TEST.ERR TIMER_TEST.C
IIC.O: IIC.C
        $(CC) $(CFLAGS)  -ERRORS IIC.ERR IIC.C
IIC_TEST.O: IIC_TEST.C
        $(CC) $(CFLAGS)  -ERRORS IIC_TEST.ERR IIC_TEST.C
SYSTEM.O: SYSTEM.C
        $(CC) $(CFLAGS)  -ERRORS SYSTEM.ERR SYSTEM.C
DMA.O: DMA.C
        $(CC) $(CFLAGS)  -ERRORS DMA.ERR DMA.C
KSLIB.O: KSLIB.C
        $(CC) $(CFLAGS)  -ERRORS KSLIB.ERR KSLIB.C
MAC.O: MAC.C
        $(CC) $(CFLAGS)  -ERRORS MAC.ERR MAC.C
MACINIT.O: MACINIT.C
        $(CC) $(CFLAGS)  -ERRORS MACINIT.ERR MACINIT.C
MACLIB.O: MACLIB.C
        $(CC) $(CFLAGS)  -ERRORS MACLIB.ERR MACLIB.C
HDLC.O: HDLC.C
        $(CC) $(CFLAGS)  -ERRORS HDLC.ERR HDLC.C
HDLCINIT.O: HDLCINIT.C
        $(CC) $(CFLAGS)  -ERRORS HDLCINIT.ERR HDLCINIT.C
HDLCLIB.O: HDLCLIB.C
        $(CC) $(CFLAGS)  -ERRORS HDLCLIB.ERR HDLCLIB.C
HDLCMAIN.O: HDLCMAIN.C
        $(CC) $(CFLAGS)  -ERRORS HDLCMAIN.ERR HDLCMAIN.C
HDLC100INIT.O: HDLC100INIT.C
        $(CC) $(CFLAGS)  -ERRORS HDLC100INIT.ERR HDLC100INIT.C
HDLC100LIB.O: HDLC100LIB.C
        $(CC) $(CFLAGS)  -ERRORS HDLC100LIB.ERR HDLC100LIB.C
IOP.O: IOP.C
        $(CC) $(CFLAGS)  -ERRORS IOP.ERR IOP.C
DHRY_1.O: DHRY_1.C
        $(CC) $(CFLAGS)  -ERRORS DHRY_1.ERR DHRY_1.C
DHRY_2.O: DHRY_2.C
        $(CC) $(CFLAGS)  -ERRORS DHRY_2.ERR DHRY_2.C


README.TXT

;/*****************************************************************************************/
;/*                                                                                       */
;/* FILE NAME                                VERSION                                      */
;/*                                                                                       */
;/*     readme.txt                                SNDS100 Board version 1.0               */
;/*                                                                                       */
;/* COMPONENT                                                                             */
;/*                                                                                       */
;/*                                                                                       */
;/* DESCRIPTION                                                                           */
;/*                                                                                       */
;/*      It'll be helpful to build Boot ROM or download DRAM image or                     */
;/*      ARM debugger image format for KS32C5000A/KS32C50100.                             */
```

```
;/*                                                                                          */
;/* AUTHOR                                                                                   */
;/*                                                                                          */
;/*                                                                                          */
;/* HISTORY                                                                                  */
;/*                                                                                          */
;/*       NAME          DATE                                    REMARKS                    */
;/*                                                                                          */
;/*       in4maker      03-10-1999              Created initial version 1.0                */
;/*                                                                                          */
;/*******************************************************************************************/
```

1. HOW TO BUILD DIAGNOSTIC IMAGE FILES FOR KS32C5000A/KS32C50100 ?

****
1) First of all, you have to change the option flags in makefile.
Option flags are as follows:
****

(1) ARM SDT ENVIRONMENT SETUP & SELECT DEVICE

Assign ARM SDT Toolkits installed directory path to ARMPATH.
And also select used device as like KS32C5000A or KS32C50100

```
+-------------------------------------------------------------------------------------+
ARMPATH =   e:\arm211a
DEVICE  =   KS32C50100
+-----------------------------------------------------------+
```

(2) C-Predefined option flags.

```
+-------------------------------------------------------------------------------------------+
+              C-Predefined option flags                    +
+-------------------------------------------------------------------------------------------+
+ [LITTLE]      This flag have to be defined for the device to          +
+              operate in little endian mode.                  +
+              HDLC,MAC block will be affected by this flag.         +
+ [EX_UCLK]   Using for external UART clock  [usage: -DEX_UCLK]   +
+ [PHY_100M]  Configure PHY to 100Mbps                  +
+ [SNDS1_2]    Have to be defined using SNDS ver1.2 board with      +
+              KS32C5000A without revise the current diagnostic      +
+              source code.                          +
+-------------------------------------------------------------------------------------------+
CPDFLAGs     = -D$(DEVICE) -DEX_UCLK
+-------------------------------------------------------------------------------------------+
```

(3) DEFINE IMAGE FORMAT AS LIKE ARM/THUBM AND BIG/LITTLE

```
+-------------------------------------------------------------------------------------------+
              THUMB(BIG/LITTLE)        ARM(BIG/LITTLE)
+-------------------------------------------------------------------------------------------+
ARMLIB      =  armlib.16b/armlib.16l  armlib.32b/armlib.32l
C           =           tcc                   armcc
ASSEMBLER   =       tasm                  armasm
INTERWORK   =      /interwork
```

```
   CPU_OPTs   =                    -processor ARM7TM -arch 4T
   +-------------------------------------------------------------+
```

(4) TO BUILD BOOT ROM IMAGE

```
   +-------------------------------------------------------------+
   IMAGE     = bin
   APDFLAG   = TRUE
   MEMORY    = -RO 0x0 -RW 0x1000050
   +-------------------------------------------------------------+
```

(5) TO BUILD DOWNLOAD DRAM IMAGE

```
   +-------------------------------------------------------------+
   IMAGE     = bin
   APDFLAG   = FLASE
   MEMORY    = -RO 0x1000050 -RW 0x13000000
   +-------------------------------------------------------------+
```

(6) TO BUILD ARM DEBUGGER IMAGE

```
   +-------------------------------------------------------------+
   IMAGE     = aif
   APDFLAG   = FLASE
   MEMORY    = -RO 0x1000050 -RW 0x13000000
   +-------------------------------------------------------------+
```

****

2) sysconf.h

If you want to change the device's mode like as UART baud rate ,IIC serial clock frequency, Internal system clock(fMCLK) etc, please update this file.

****

3) snds.a

SNDS100 Board memory configurations and memory MAP can be changed by this file.

Where, the fMCLK also have to be updated to same value as fMCLK defined at sysconf.h file.

******** End of readme.txt **********

Now, you have created makefile to build image for your target systems. And then, you have to compile and link using this file by armmake in DOS windows. *First of all, you have to configure the ARM Project Manager environments(refer to USING ARM PROJECT MANAGER TO BUILD IMAGE) because its default option values are effects on when run the armmake with makefile* .

Its ready to build image in DOS windows . There are four cases as follows:

1. To build 32bit ARM image for KS32C5000/5000A,

   Armmake diag

2. To build 16bit Thumb image for KS32C5000/5000A.

   Armmake tdiag

3. To build 32bit ARM image for KS32C50100,

   Armmake diag100

4. To build 16bit Thumb image for KS32C50100.

   Armmake tdiag100

After this is done with no compile and link error, the following files will be generated at your working directory.

Diag /diag100/tdiag/tdiag100	:	Executable image file which will be ARM or binary image format.
Diag.sym	:	symbol file
<file_name>.o	:	Object file
<file_name>.err	:	Error and warning list files.

If you have emulator (Ex : EmbeddedICE) or Angel potted ROM, you can use ARM debugger window(ADW) .If not, you can download executable binary image file to targets by sftp.exe which is ours program tips. Please refer to the next paragraph.

DOWNLOADING EXECUTABLE BINARY IMAGE FILE WITHOUT ADW

Without a Emulator(ex, EmbeddedICE), you can download a binary image file through the serial cable to target. First of all, you ought to get download utility, sftp.exe, from our web sites .

To download a executable binary file, following the steps:

1. Select User Pgm to download item on the SNDS100 console(Hyper-Terminal) menu .
   It is referred to Figure 2-22.

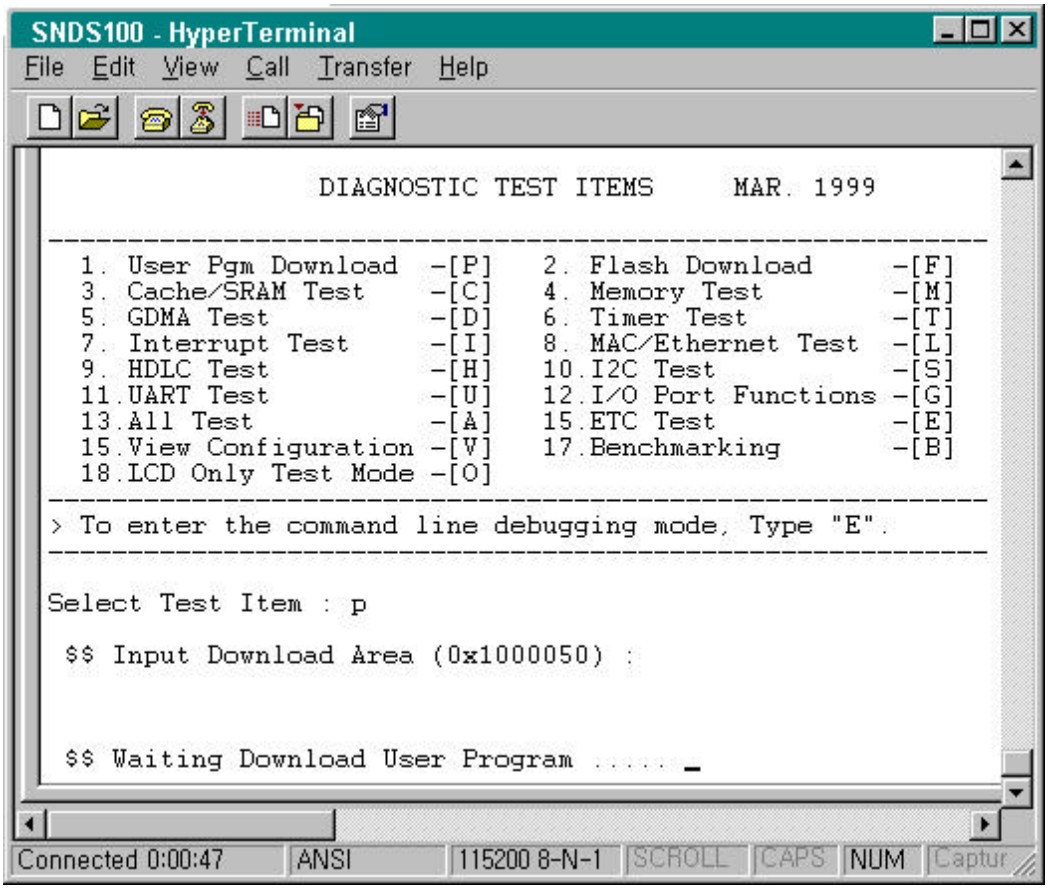


Figure 2-22. Download Executable Binary Image on SNDS100 Board

DOWNLOADING EXECUTABLE BINARY IMAGE FILE WITHOUT ADW (Continued)

2. Start download the binary image file on host computer.(refer to Figure 2-21)

   `sftp.exe` is used to download binary image file to target. This execution file is compiled at Window95, and the compiler is `Visual C++` If you are in a different environment, you should recompile the serial download utility source code in your environments. More detail information about `sftp.exe` is described later this chapter.

   If you already have the utility tips, Now, you can send the image file from COM1 or COM2 port on Host PC through the serial cable to Console port(Default, UART channel 0, SIO-0) on SNDS100. For example, if the UART baud rate is 125200 on SNDS100, The serial cable is connected from COM1 to SIO-0, then you have to type `*DOS> sftp 1 diag100*` *on DOS window*`s command Prompt.

   USAGE : SFTP <COM PORT> <FILE NAME>

   Example : dos> sftp 1 diag100

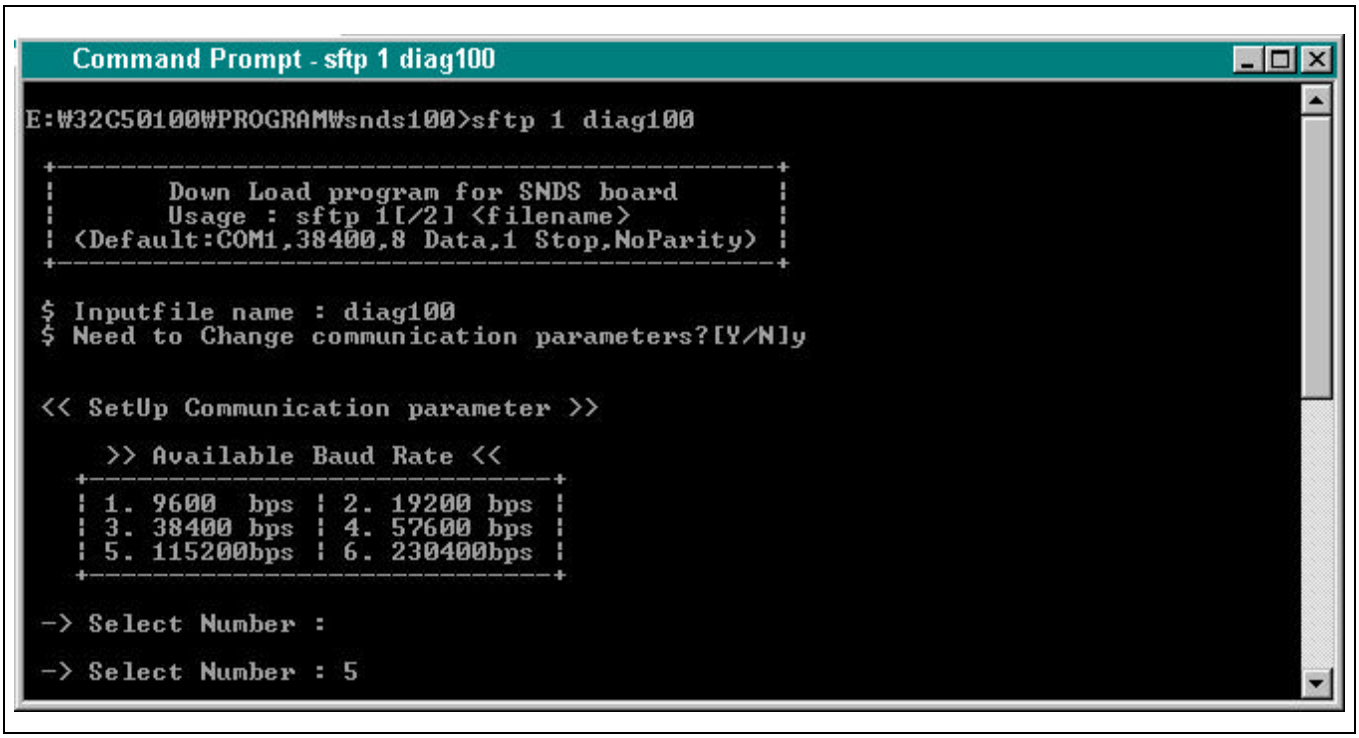


Figure 2-21. Start Download Binary Image File to Target on Host Computer

2. When the download is finished without any error, press any key to start the download user binary image file on SNDS100 target board.

Now, you can simply evaluating and verifying the downloaded applications on SNDS100.

FUSING BOOT ROM IMAGE TO FLASH(EEPROM)

You can update the boot ROM on SNDS100 into your new boot image include applications using flash download program which is included in diagnostic code. (please, visit our web site to download this source codes).

SST flash device used as Boot EEPROM on SNDS100 board. So, the flash program in diagnostic code is for the SST device. If you want to use any other vendor' s flash device, you have to programming for that to fuse the application to Flash memory without ROM writer.

The above case is just only for updating boot ROM on SNDS100 board. If you have an Emulator(EmbeddedICE), you can fuse the boot & applications through the JTAG port to flash device on board without boot ROM. In this case, the SNDS100 memory map of the CPU(KS32C5000/5000A/50100) have to be configured by ADW. More detail will be described in this section.

UPDATE BOOT ROM ON SNDS100 BOARD

First of all, you have to prepare the DRAM image file which will be used to fuse new boot ROM image or applications to flash memory on SNDS100 board. To build ROM or DRAM image, please refer to "USING MAKEFILE FOR IMAGE BUILD" section of this chapter.

On SNDS100 board, the bus width of Boot ROM can be extended to 16bit. Actually, the bus width of ROM Bank0(Boot ROM bank) can be configured by B0SIZE0(pin 73) and B0SIZE1(pin 74), but you have to edit the flash.h file to modify the bus width of ROM Bank0 as same value as it.

Detail flow charts for fusing EEPROM are given here. And also the procedure according to that is following:

1. Open flash.h file for to update the bus width of ROM Bank0(Boot ROM).

   For example, if you want to use 16bit Flash device, then modify the define statement to "#define *B0SIZE B0SIZE*_SHORT" in flash.h file.

2. Build DRAM image file.

   Please refer to "USING MAKEFILE FOR IMAGE BUILD" section for more detail DRAM image build.

3. Download DRAM image.

   Firstly, you have to *enter a character `P`* to select user pgm download item on Console terminal (Hyper Terminal). And then you can send DRAM image file to target board on DOS prompt.

   DOS> sftp 1 diag100

4. Execute download image.

   After the DRAM image is downloaded to DRAM successfully, press any key to restart the target board. Now, the downloaded DRAM program is running on DRAM area. Next, *you have to enter a character `F`* to download the new Boot ROM image or Applications to DRAM user area.

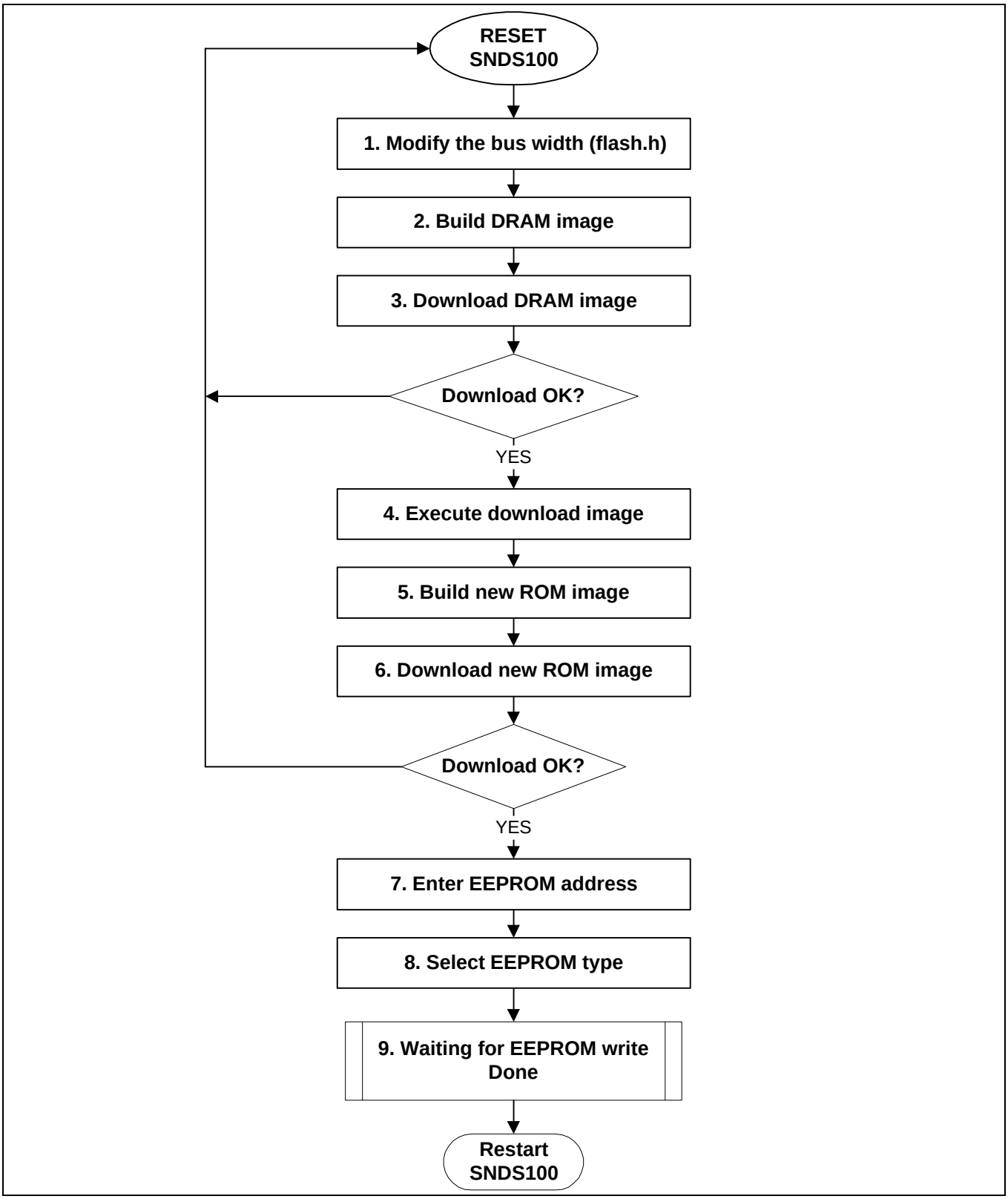


Figure 2-22. Fusing Flow to Update Boot ROM

UPDATE BOOT ROM ON SNDS100 BOARD (Continued)

5. Build New ROM image.

   This image is for the new boot code or applications. It also refer to `USING MAKEFILE FOR IMAGE BUILD` section.

6. Download New ROM image.

   This new ROM image or Applications will be downloaded to DRAM user area and also fused to flash device by flash write program running on DRAM.

7. Enter EEPROM address.(Figure 2-23)

   If there is no error for downloading, the console (Hyper Terminal) request to enter the EEPROM load address which is default zero. After this, SELECT EEPROM TYPE menu is displayed on Console window.

8. Select EEPROM type.(Figure 2-23)

9. Waiting for EEPROM write is DONE.

10. Now, press RESET button on SNDS100 target board to restart it.

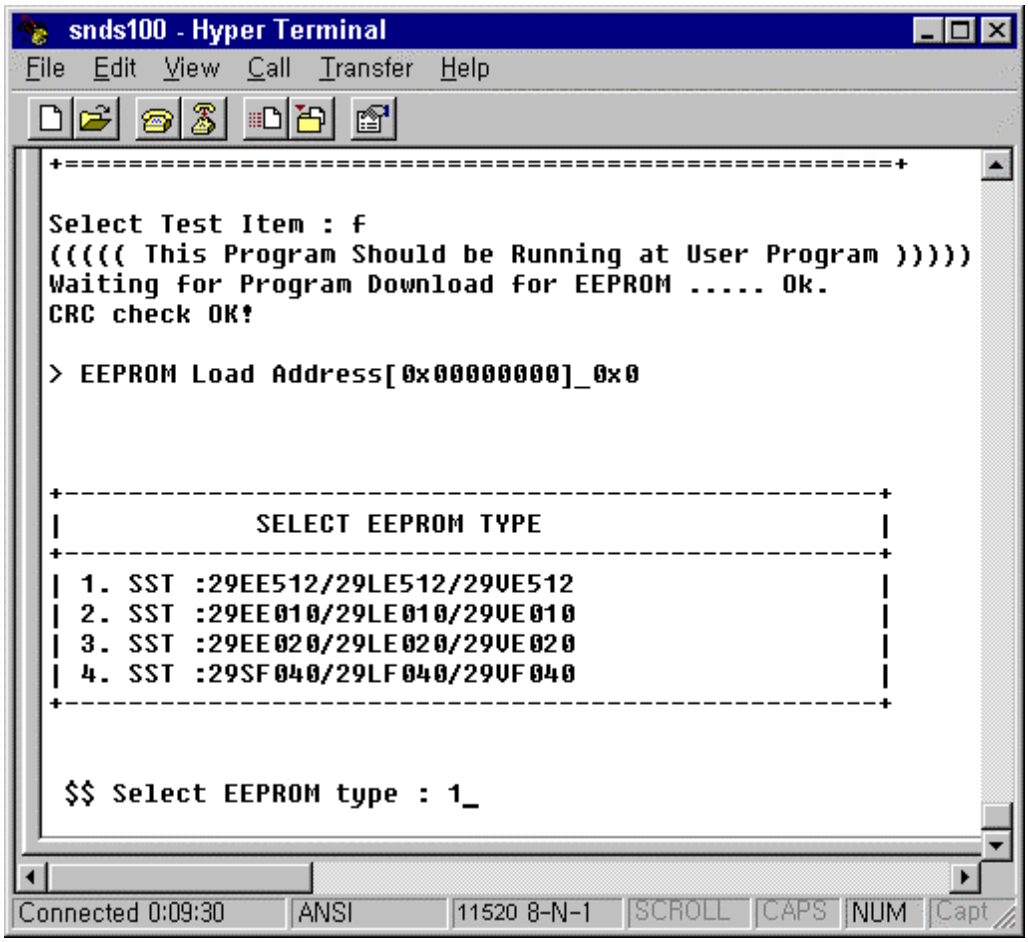


Figure 2-23. Start Flash Download Program

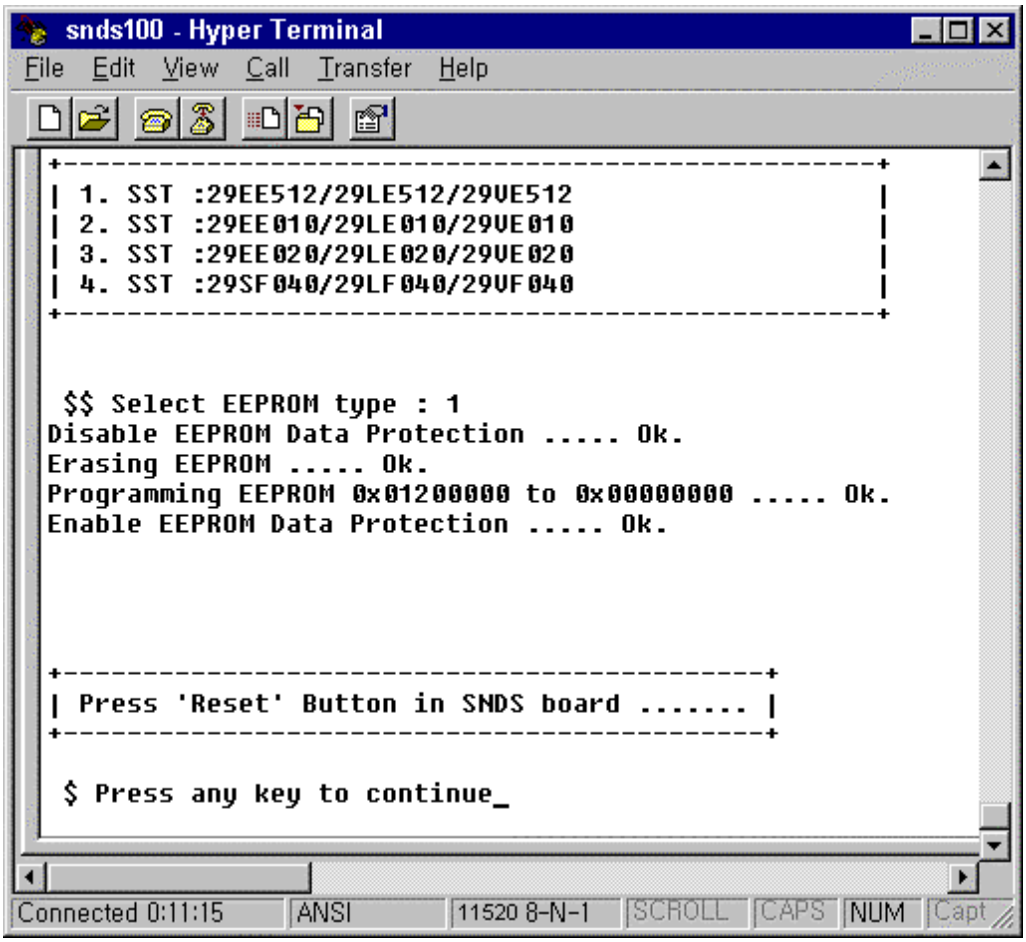


Figure 2-24. Finished Flash Download Program

SERIAL DOWNLOAD PROGRAM ON HOST COMPUTER

We provide the download utility for DOS. This program tips which are compressed into SERIAL.zip file are available on our web site, (www.samsungsemi.com). In this zip file, the "Serial.c" which is main source code is for to manipulate binary data , and the "Trans.c" for to manipulate the serial communications of host computer.

We compiled this source code by "Visual C++" and generated execution file, sftp.exe. Using this download utility tips, you can download executable binary image for ROM, DRAM to SNDS100 target board.

This utility tips is just only for DOS. So, if you want to send binary image file to target, you have to run this tips on DOS Prompt. It' s usage is " DOS> sftp <COM1/COM2> <Binary Image file>".

BINARY DATA TRANSFER FORMAT

To transfer a file to target, the serial transfer utility program for host computer and the flash down program in diagnostic code for SNDS100 use any transfer format which is shown in Figure 2-25.

Length (8 bytes)	DATA (variable)	CRC (8 bytes)

Figure 2-25. Binary Data Transfer Format

Where the Length field is only data(download file) size. Length and CRC field is calculated by download utility program.

CRC CHECK FOR ERROR CHECK

We use CRC check method for error check, The CRC check use CCITT polynomial, and we check all 8-bit data for each character.

Code for CRC Calculation

```
for( j = 0; j < 8; j++ )
{
  CheckSum <<= 1;
  if( CharData & 0x0080 ) CheckSum ^= CCITT_POLYNOM ;
  CharData <<= 1;
 }
```

DATA TRANSFER FLOW

File transfer flows are shown in Figure 2-26. Where the data(File) is an executable binary ROM or DRAM image or general text file.

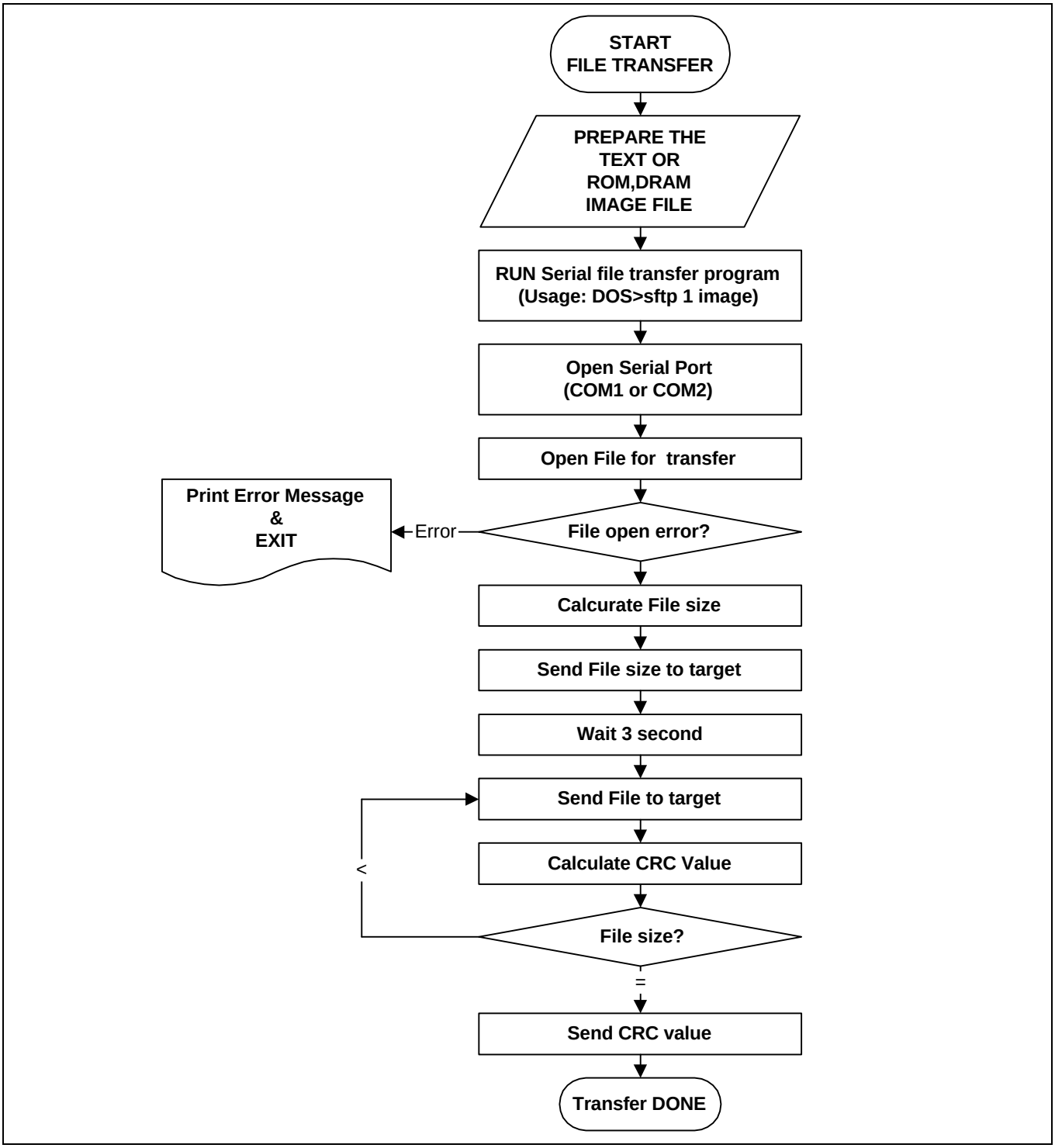


Figure 2-26. File Transfer flow

3 DIAGNOSTIC SOURCE CODES

THE SNDS100 PROGRAMMER' S MODEL

The diagnostic code routines have been written to give practical examples of writing code and evaluating SNDS100 (Samsung Network Development System) board. SNDS100 is the target board of the NetMCU-I series like as KS32C5000/5000A/50100 which are 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 SNDS100 board.

HARDWARE OVERVIEW

— BOOT ROM: 16bit data bus, 1Mbit EEPROM *2, extent to 4Mbit *2, Located at ROM Bank0

— DRAM TYPE: Normal/EDO DRAM for KS32C5000/5000A, Normal/EDO or SyncDRAM for KS32C50100. DRAM Bank0 used.

— CONSOLE: UART CHANNEL 0(SIO-0) used.

— ETHERNET: RJ45 connector, MII interface, 10/100Mbps can be configured by auto-negotiation.

— JTAG: Embedded ICE or Emulator can be interfaced with this for system debuging.

SYSTEM MEMORY MAP

The SNDS100 board has 2 ROM sockets which can be used to boot ROM. It' s data bus size can be configured to 8bit or 16bit by the B0SIZE[1:0] pins which can be controlled by switch.

For the code development, SNDS100 board support SyncDRAM into component type and also one DRAM SIMM sockets. There are all located on DRAM Bank 0. It' s means that two DRAM memory have to be selected alternatively according to the kind of attached devices. KS32C5000/5000A only support Normal/EDO DRAM so that the DRAM SIMM only available for it. In case of KS32C50100, both memory type can supported. So you can select DRAM SIMM or SyncDRAM by jumper on SNDS100.

The NetMCU-I have 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, the base address of the internal SRAM, and also control the write buffer, cache, stall, cache mode. The specatial register' s address area is fixed at 64Kbytes. It' s initial value is 0x3FFFF91.

You can use the internal 8-Kbytes SRAM as a cache using Cache control bit in SYSCFG register. 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 [21] of memory access address.

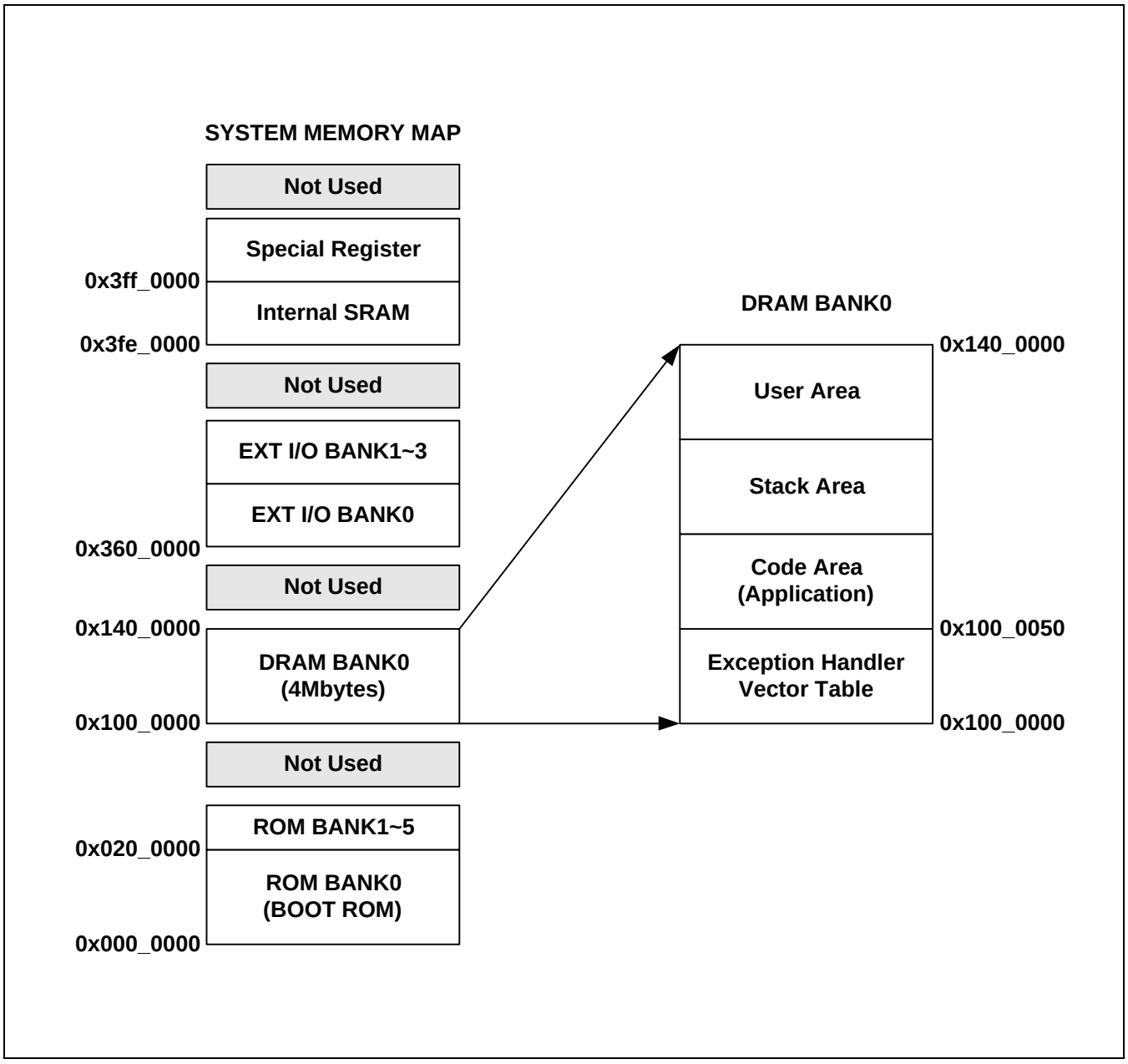


Figure 3-1. SNDS100 System Memory MAP

DATA STRUCTURES OF DRAM BANK0

The SNDS100 system memory allocation table is configured by the compiler. Figure 3-2 shows the data structures of DRAM bank0.

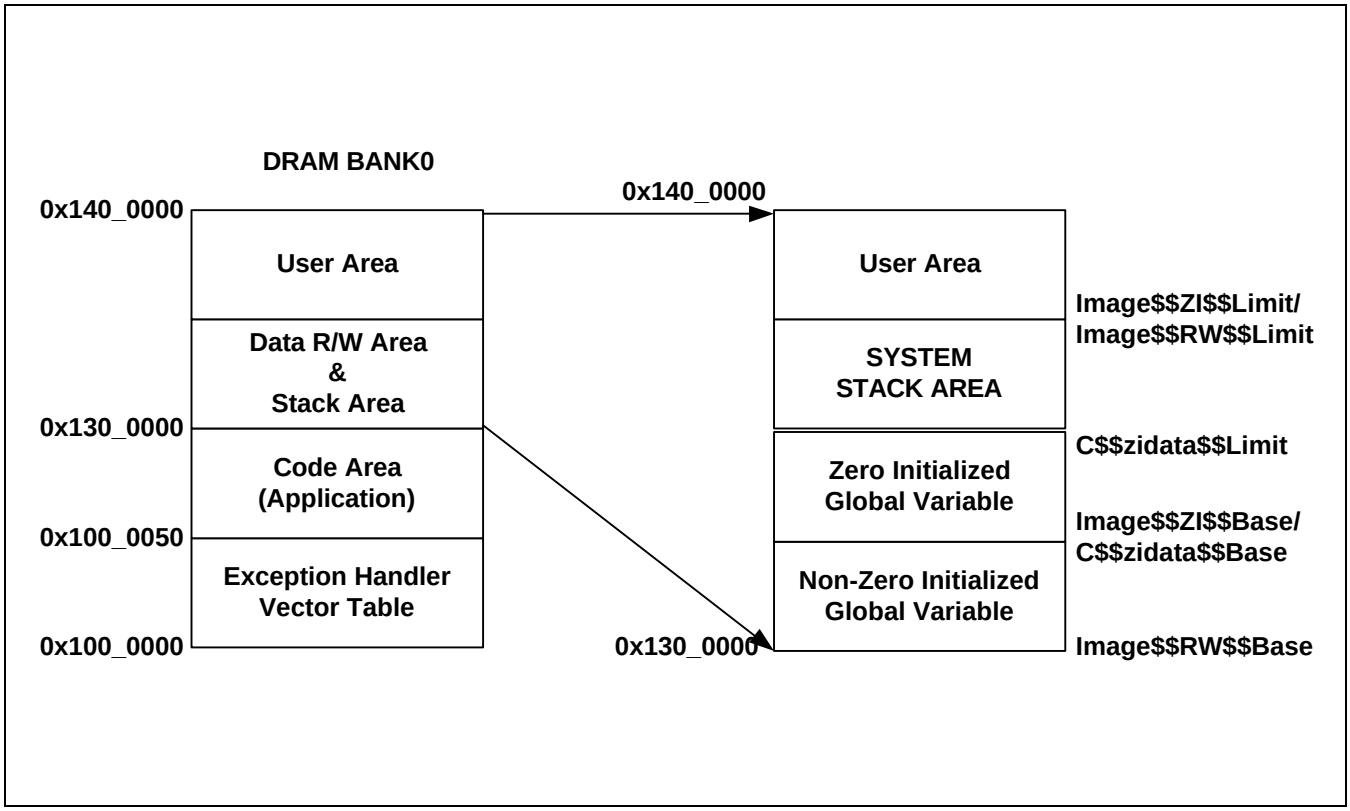


Figure 3-2. Data Structures of DRAM Bank0

BOOT PROCEDURES

After a power on reset or a reset signal input, the NetMCU device on SNDS100 access the ROM located at address 0x0. This ROM is written to boot code initializing the system configuration and executing a diagnostic code for evaluation.

Figure 3-3 shows the boot processing flow by boot ROM. The start-up code for booting process is programed by assembler code and the diagnostic by C-Source code.

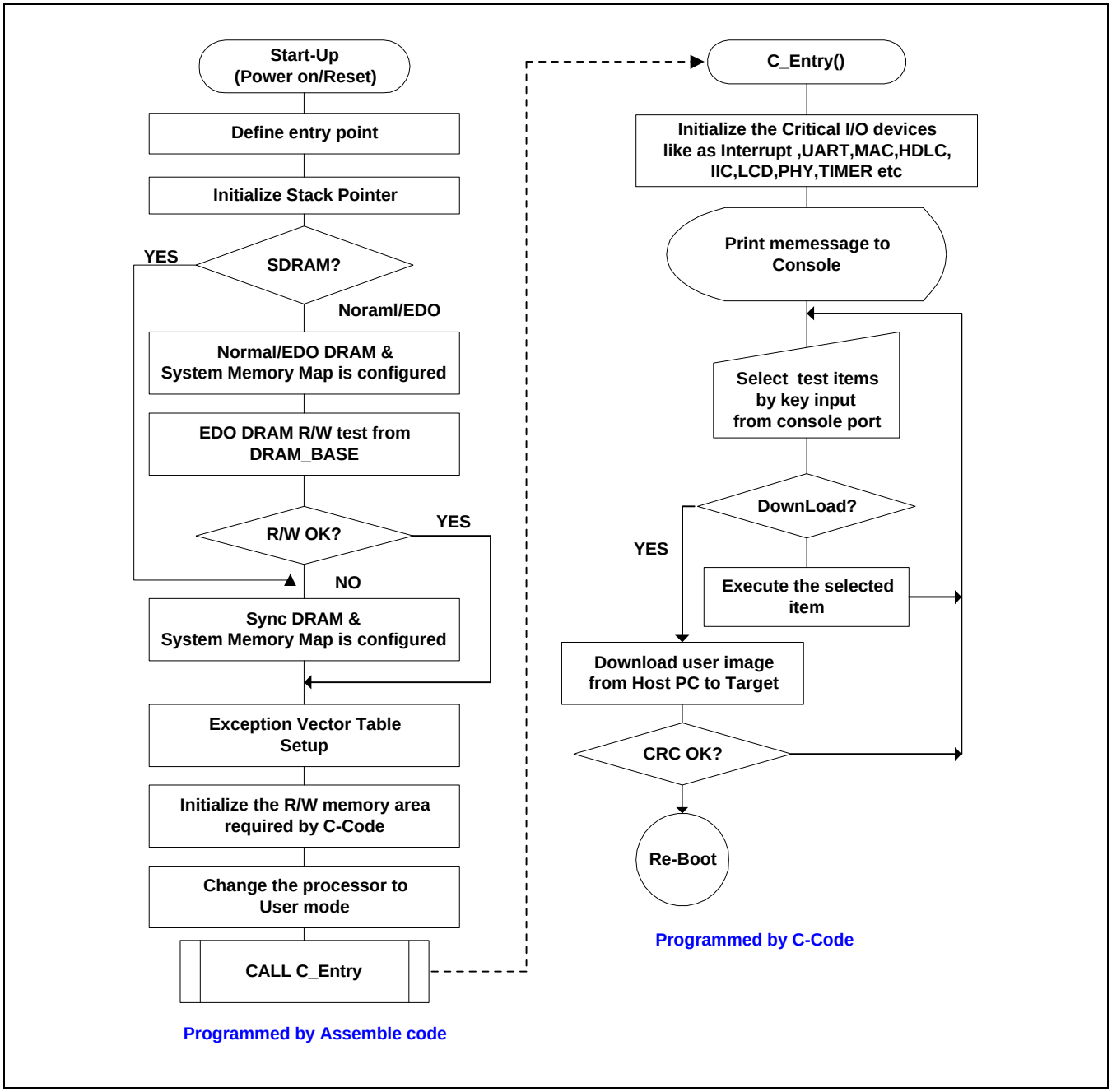


Figure 3-3. The Boot processing flows of Boot ROM.

EXCEPTION HANDLING

NetMCU devices have seven type of exceptions as shown in Table 3-2. More details about exceptions refer to the document, "*ARM Software Development Toolkit User Guide, Section 10*".

If an exception occurs, NetMCU devices act on it for handling which flows are described at Figure 3-4.

Table 3-2. Exception Handling Priorities

Exceptions	Descriptions	Vector Address	Priority
RESET	CPU Reset input or Power on Reset	0x00	1
Undefined	Undefined instruction	0x04	6
SWI	User-Defined synchronous interrupt	0x08	6
Prefetch abort	Prefetch from illegitimate address	0x0C	5
Data abort	Data load/store at an illegitimate address	0x10	2
IRQ	Normal interrupt	0x18	4
FIQ	Fast interrupt	0x1C	3

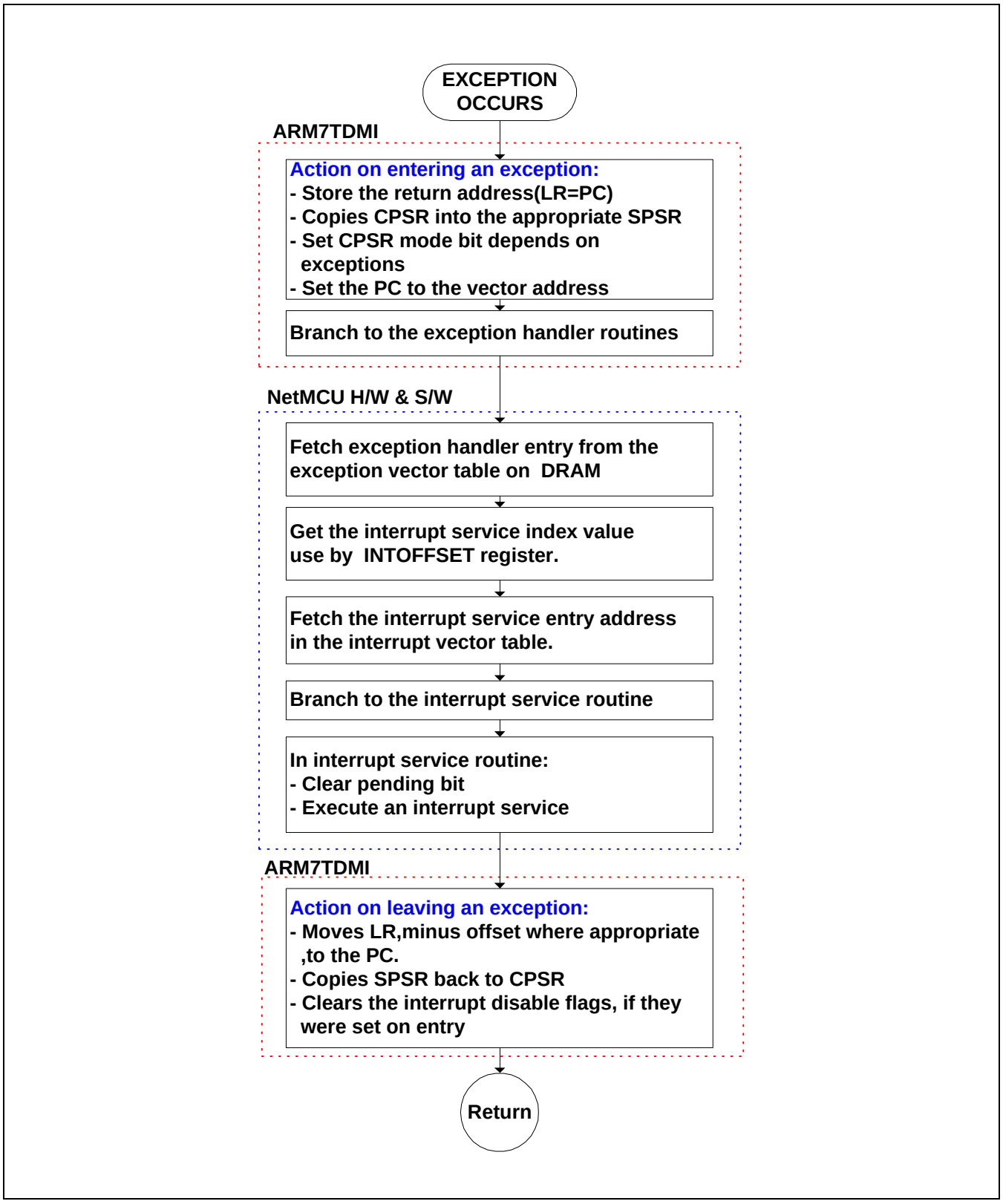


Figure 3-4. Exception handling flows of NetMCU devices

STRUCTURES OF DIAGNOSTIC CODES

Table 3-1 shows the structures of diagnostic source codes. These sources are useful for evaluate SNDS100 target Board. And also, it will be helpful for you to understand our NetMCU devices which have many embeded control block.

Table 3-1. Diagnostic code structure

```
/* boot codes  */

Init                              ; init.s, snds.a, memory.a, snds.h

/* Diagnostic source codes for evaluating the target board */

C_Entry()                         ; diag.c, diag.h, kslib.c, kslib.h, sysconf.h
  |--------- MemTestDiag()        ; memory.c, memory.h
  |--------- CacheTestDiag()      ; system.c, system.h
  |--------- DmaTestDiag()        ; dma.c, dma.h
  |-------- TimerTestDiag()       ; timer.c, timer.h,timer_test.c
  |-------- MacTestDiag()         ; mac.c, mac.h, macinit.c, maclib.c
  |-------- HdlcTestDiag()        ; hdlc.c, hdlc.h, hdlcinit.c, hdlclib.c  for KS32C5000/5000A
  |                   ; hdlcmain.h, hdlcmain.c, hdlc100init.c, hdlc100lib.c for 50100
  |-------- I2CtestDiag()         ; iic.c, iic.h, iic_test.c
  |-------- UartTestDiag()        ; uart.c, uart.h, uart_test.c
  |-------- IoFuncDiag()          ; iop.c, iop.h
  |-------- DownLoadDiag()        ; down.c, down.h
  |-------- FusingDiag()          ; flash.c, flash.h
  |-------- LcdOnlyTestMode()     ; lcd.c, lcd.h
  |-------- BanchMarkDiag()       ; dhry_1.c, dhry_2.c, dhry.h
```

BOOT CODES

Memory.a SNDS100 board system memory map

Snds.a Assembler code header file for NetMCU-I(KS32C5000/5000A/50100). Defined EQU table In this file will be helpful for you to configure and initialize the system memory map.

Snds.h C- header file of NetMCU-I for SNDS100 target board. System registers are defined in this file.

Init.s Assemble code for the booting procedure. It performs the stack initialization, system memory Configuration, exception vector table setup and C required memory initailiztion. After this boot procedure is done, It branched to C_Entry point to run appliations.

DIAGNOSTIC C-SOURCE CODES

sysconf.h,	System configuration header file. This file include the peripheral control parameters like as UART baudrate, IIC serial frequency, Operating frequency etc,. While you evaluate the target board, you want to change the above parameters frequently
system.c, system.h	This example C source code shows the usage of cache and internal SRAM. CPU clock control using the clock control register is also evaluated.
dma.c, dma.h	Data transfer Memory to memory, UART to memory, memory to UART .
timer.c, timer.h	Usage for timer functions.
mac.c ,mac.h **macinit.c, maclib.c**	There are many functions for receiving and transmitting ethernet frames such as rx/tx descriptors, station management, etc. External and internal loopback programs are also included here for the perpose of diagnostic test the MAC control block.
hdlc.c, hdlc.h **Hdlcinit.c,hdlclib.c**	Diagnostic code for KS32C5000/5000A HDLC (High-level Data Link Control) control functions.
hdlcmain.c, **hdlcmain.h** **Hdlc100init.c**, **hdlc100lib.c**	Diagnostic code for KS32C50100 HDLC (High-level Data Link Control) control functions.
iic.c, iic.h	To evaluate the IIC-bus control block, an Samsung IIC serial EEPROM, KS24C64 were used.
uart.c, uart.h **pollio.c**	External loop back using interrupt mode and internal loop back using polling mode codes are described here. There are also some libraries for standard input through UART such as get_byte, put_byte,put_string, Print etc.
iop.c, iop.h	I/O ports input/output and shared functions such as external interrupt, external DMA request and acknowlege, timer-out put test codes are described here.
lcd.c, lcd.h	SNDS100 support the LCD display pannel. Many LCD display functions are implemented. SNDS100 target board can be evaluated at stand alone using LCD display as console and Key pad on target without a connection from the host PC to UART .
dhry_1.c, dhry_2.c **dhry.h**	This is a benchmarking program that measure the system performance (dhrystone VAX mips).

down.c, down.h, start.s This code includes functions for dowmloading user program through UART channel from host PC. Downloaded user program will be located in a given area on DRAM.

flash.c, flash.h After the code debugging is finished, applications can be written to EEPROM on the SNDS100 board directly by this program. These codes are only for the SST EEPROM. For using anyother vendor' s EEPROM, you yourself have to prepare the EEPROM write Program.

PROGRAM TIPS FOR DOWNLOAD UTILITY (FOR DOS)

serial.c, serial.h, tran.c Compile this tips using Visual C++ 4.0. After that you can run this tips on DOS window to transfer image file to target from the serial port (COM1/COM2) of the Host PC through serial cable to UART channel on SNDS100 board.

CODE DESCRIPTIONS FOR EACH FUNCTION MODULES

INITIAL START-UP(BOOT) CODE

“Init.s” is assembler code for boot process. System memory map configurations, Stack initialization, Exception vector table generation, C-required memory initialization can be done by this start-up code. Also "ARM Software Development Toolkit User Guide" can be referenced to help writing a Code to ROM.

Figure 3-1,3-2 shows the memory map & usage of the SNDS100 board.

Listing 3-1. (init.s)

```
; --- Define entry point
        EXPORT  __main  ; defined to ensure that C runtime system
__main                  ; is not linked in
        ENTRY
```

When the compiler compiles the function main(), it generates a reference to the symbol __main to force the linker to include the basic C runtime system from the semi-hosted C library. If the semi-hosted C library is not linked while having the main() function, an error may occur.

To avoid an confusion, you are advised to call something other than main(), such as C_Entry() or ROM_Entry() as the C entry point, when building a ROM image

Listing 3-2.(init.s)

```
; --- Setup interrupt / exception vectors
    IF :DEF: ROM_AT_ADDRESS_ZERO
; If the ROM is at address 0 this is just a sequence of branches
        B       Reset_Handler
        B       Undefined_Handler
        B       SWI_Handler
        B       Prefetch_Handler
        B       Abort_Handler
        NOP                     ; Reserved vector
        B       IRQ_Handler
        B       FIQ_Handler
    ELSE
; Otherwise we copy a sequence of LDR PC instructions over the vectors
; (Note: We copy LDR PC instructions because branch instructions
; could not simply be copied, the offset in the branch instruction
; would have to be modified so that it branched into ROM. Also, a
; branch instructions might not reach if the ROM is at an address
; > 32M).
        MOV     R8, #0
        ADR     R9, Vector_Init_Block
        LDMIA   R9!, {R0-R7}
        STMIA   R8!, {R0-R7}
        LDMIA   R9!, {R0-R7}
```

```
        STMIA   R8!, {R0-R7}

; Now fall into the LDR PC, Reset_Addr instruction which will continue
; execution at 'Reset_Handler'

Vector_Init_Block
        LDR     PC, Reset_Addr
        LDR     PC, Undefined_Addr
        LDR     PC, SWI_Addr
        LDR     PC, Prefetch_Addr
        LDR     PC, Abort_Addr
        NOP
        LDR     PC, IRQ_Addr
        LDR     PC, FIQ_Addr

Reset_Addr      DCD     Reset_Handler
Undefined_Addr  DCD     Undefined_Handler
SWI_Addr        DCD     SWI_Handler
Prefetch_Addr   DCD     Prefetch_Handler
Abort_Addr      DCD     Abort_Handler
                DCD     0       ; Reserved vector
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler
    ENDIF


;=========================================================
; 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}
```

Listing 3-3. (memory.a)

```
;/*************************************************************************/
;/*  DRAM Memory Bank 0 area MAP for Exception vector table */
;/*  and Stack, User code area.                          */
;/*************************************************************************/
DRAM_BASE       EQU     0x1000000
;DRAM_LIMIT     EQU     0x1800000
DRAM_LIMIT      EQU     0x1400000
;-------------------------------------------------
ExceptionSize   EQU     0x50            ; exception vector addr pointer space
SysStackSize    EQU     1024 * 8        ; Define 8K System stack
ExceptionTable  EQU     DRAM_BASE + ExceptionSize

;/* EXCEPTION HANDLER VECTOR TABLE */

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

```
;/* SYSTEM USER CODE AREA  */

        ^       DRAM_BASE+ExceptionSize     ;=0x1000050
UserCodeArea        #4
```

Locating the boot ROM at address zero, The Pre-Defined symbol, ROM_AT_ADDRESS_ZERO have to be assigned to logical value TRUE.

An application image with boot procedure can be downloaded and run on DRAM as assign logical value FALSE to ROM_AT_ADDRESS_ZERO on debugging stage. In this case, the boot sequence of download image is not located at address zero so that the exception vector table on boot ROM have to be linked to exception vector table on DRAM .

At the above boot sequence shows that the both case whether boot code located at address zero or not are same as that have exception vector table located on DRAM region. It will be convenient for you to debug an application and build to ROM image with same code. But, it also caused to be a latency problem handling the exceptions.

To avoid the exception latency problems and slow execution of application on ROM, ROM image can be copied to DRAM and change the memory map to locate the DRAM at address zero at power on reset booting procedures. More details will be described in this section.

Listing 3-4.(init.s)

```
        AREA Main, CODE, READONLY

;=========================================================
; The Reset Entry Point
;=========================================================
Reset_Handler                   ;/* Reset Entry Point */

 [ ROM_AT_ADDRESS_ZERO
 |
        LDR     r0, =HandleSwi          ; SWI exception table address
        LDR     r1, =SystemSwiHandler
        STR     r1, [r0]
        swi 0xff                                ;/* Call SWI Vector  */
 ]
```

The code(Listing 3-4) Is for the boot processing of download image. After the boot sequence of boot ROM is done, The CPU status is in user mode. The boot processing can be made only in supervisor mode. So, to execute the boot sequence by the download application, CPU mode have to changed to supervisor mode using call SWI handler.

Listing 3-5.(init.s)

```
        ;===================================
         ; Initialize STACK pointer
         ;===================================
INITIALIZE_STACK
        MRS     r0, cpsr
        BIC     r0, r0, #LOCKOUT | MODE_MASK
        ORR     r2, r0, #USR_MODE

        ORR     r1, r0, #LOCKOUT | FIQ_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =FIQ_STACK

        ORR     r1, r0, #LOCKOUT | IRQ_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =IRQ_STACK

        ORR     r1, r0, #LOCKOUT | ABT_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =ABT_STACK

        ORR     r1, r0, #LOCKOUT | UDF_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =UDF_STACK

        ORR     r1, r0, #LOCKOUT | SUP_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =SUP_STACK   ; Change CPSR to SVC mode
```

Listing 3-6. (init.s)

```
;/*************************************************/
        AREA SYS_STACK, NOINIT
;/*************************************************/
        %       USR_STACK_SIZE
USR_STACK
        %       UDF_STACK_SIZE
UDF_STACK
        %       ABT_STACK_SIZE
ABT_STACK
        %       IRQ_STACK_SIZE
IRQ_STACK
        %       FIQ_STACK_SIZE
FIQ_STACK
        %       SUP_STACK_SIZE
SUP_STACK

;/*************************************************/
        END
```

System Stack will be allocated to Non-initialized R/W area of DRAM by ARM linker when the image is build.

Listing 3-7. (init.s)

```
        ;=====================================
        ; Setup Special Register
        ;=====================================
        LDR     r0, =0x3FF0000              ; Read SYSCFG register value
        LDR     r1,[r0]                     ; To identify DRAM type
        LDR     r2, =0x80000000
        AND     r0, r1, r2                  ; Mask DRAM type mode bit
        CMP     r0, r2
        BNE     EDO_DRAM_CONFIGURATION
        B       SYNC_DRAM_CONFIGURATION  ; only when KS32C50100

        ;================================================
        ; Special Register Configuration for EDO mode DRAM
        ; When KS32C5000 and KS32C50100
        ;================================================
EDO_DRAM_CONFIGURATION
        LDR     r0, =0x3FF0000
        LDR     r1, =0x3FFFF90              ; SetValue = 0x3FFFF91
        STR     r1, [r0]                    ; Cache,WB disable
                                    ; Start_addr = 0x3FF00000
        ;ROM and RAM Configuration(Multiple Load and Store)
        ADRL    r0, SystemInitData
        LDMIA   r0, {r1-r12}
        LDR     r0, =0x3FF0000 + 0x3010     ; ROMCntr Offset : 0x3010
        STMIA   r0, {r1-r12}
```

```
        LDR     r1,=DRAM_BASE
        STR     r1,[r1]                 ; [DRAM_BASE] = DRAM_BASE
        LDR     r2,[r1]                   ; Read DRAM Data
        CMP     r2,r1
        BEQ     EXCEPTION_VECTOR_TABLE_SETUP


    ;=================================================
        ; Special Register Configuration for SYNC DRAM
        ; Only when KS32C50100
        ;=================================================
SYNC_DRAM_CONFIGURATION
        LDR     r0, =0x3FF0000
        LDR     r1, =0x83FFFF90         ; SetValue = 0x83FFFF91
        STR     r1, [r0]                ; Cache,WB disable
                                        ; Start_addr = 0x3FF00000

        ;ROM and RAM Configuration(Multiple Load and Store)
        ADRL    r0, SystemInitDataSDRAM
        LDMIA   r0, {r1-r12}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCntr Offset : 0x3010
        STMIA   r0, {r1-r12}
```

KS32C50100 support SDRAM in addition to Normal/EDO DRAM. Both memory types are mounted on the SNDS100 target board so that you can choose the memory types alternatively by set of jumper. (JP1,JP2).

After the jumper setting and power on reset, the mounted memory type can be decided by the above routine checking the memory mode bit of SYSCFG register or testing the default configured memory at first.

Now, The system stack initialization and system memory map configuration is done.

Listing 3-8.(init.s)

```
          ;============================
        ; Exception Vector Table Setup
        ;============================
EXCEPTION_VECTOR_TABLE_SETUP
        LDR     r0, =HandleReset                ; Exception Vector Table Memory Loc.
        LDR     r1, =ExceptionHandlerTable      ; Exception Handler Assign
        MOV     r2, #8                          ; Number of Exception is 8
ExceptLoop
        LDR     r3, [r1], #4
        STR     r3, [r0], #4
        SUBS    r2, r2, #1                      ; Down Count
        BNE     ExceptLoop
```

The C-code exception handlers will be setup to exception vector table on DRAM using the table of ExceptionHandlerTable.

Listing 3-9. (init.s)

```
;=======================================
; Exception Vector Function Definition
; Consist of function Call from C-Program.
;=======================================
SystemUndefinedHandler
        IMPORT          ISR_UndefHandler
        STMFD sp!, {r0-r12}
        B       ISR_UndefHandler
        LDMFD sp!, {r0-r12, pc}^

SystemSwiHandler
        STMFD sp!, {r0-r12,lr}
        LDR     r0, [lr, #-4]
        BIC     r0, r0, #0xff000000
        CMP     r0, #0xff
        BEQ     MakeSVC
        LDMFD sp!, {r0-r12, pc}^
MakeSVC
        MRS     r1, spsr
        BIC     r1, r1, #MODE_MASK
        ORR     r2, r1, #SUP_MODE
        MSR     spsr, r2
        LDMFD sp!, {r0-r12, pc}^

SystemPrefetchHandler
        IMPORT          ISR_PrefetchHandler
        STMFD sp!, {r0-r12, lr}
        B       ISR_PrefetchHandler
        LDMFD sp!, {r0-r12, lr}
        ;ADD    sp, sp, #4
        SUBS    pc, lr, #4

SystemAbortHandler
        IMPORT          ISR_AbortHandler
        STMFD sp!, {r0-r12, lr}
        B       ISR_AbortHandler
        LDMFD sp!, {r0-r12, lr}
        ;ADD    sp, sp, #4
        SUBS    pc, lr, #8

SystemReserv
        SUBS    pc, lr, #4

SystemIrqHandler
        IMPORT          ISR_IrqHandler
        STMFD sp!, {r0-r12, lr}
        BL      ISR_IrqHandler
        LDMFD sp!, {r0-r12, lr}
        SUBS    pc, lr, #4
```

```
SystemFiqHandler
        IMPORT          ISR_FiqHandler
        STMFD sp!, {r0-r7, lr}
        BL      ISR_FiqHandler
        LDMFD sp!, {r0-r7, lr}
        SUBS    pc, lr, #4
```

System exception handlers are consist of function call from C-code(**isr.c**). This handler function can be referenced by ExceptionHandlerTable.

Listing 3-10. (init.s)

```
        AREA ROMDATA, DATA, READONLY

;=====================================================
; DRAM System Initialize Data(KS32C5000 and KS32C50100)
;=====================================================
SystemInitData
        DCD rEXTDBWTH          ; DRAM1(Half), ROM5(Byte), ROM1(Half), else 32bit
        DCD rROMCON0           ; 0x0000000 ~ 0x01FFFFF, ROM0,4Mbit,2cycle
        DCD rROMCON1           ;
        DCD rROMCON2           ; 0x0400000 ~ 0x05FFFFF, ROM2
        DCD rROMCON3           ; 0x0600000 ~ 0x07FFFFF, ROM3
        DCD rROMCON4           ; 0x0800000 ~ 0x09FFFFF, ROM4
        DCD rROMCON5           ;
        DCD rDRAMCON0          ; 0x1000000 ~ 0x13FFFFF, DRAM0 4M,
        DCD rDRAMCON1          ; 0x1400000 ~ 0x17FFFFF, DRAM1 4M,
        DCD rDRAMCON2          ; 0x1800000 ~ 0x1EFFFFF, DRAM2 16M
        DCD rDRAMCON3          ; 0x1C00000 ~ 0x1FFFFFF
        DCD rREFEXTCON         ; External I/O, Refresh

;=====================================================
; SDRAM System Initialize Data  (KS32C50100 only)
;=====================================================
SystemInitDataSDRAM
        DCD rEXTDBWTH          ; DRAM1(Half), ROM5(Byte), ROM1(Half), else 32bit
        DCD rROMCON0           ; 0x0000000 ~ 0x01FFFFF, ROM0,4Mbit,2cycle
        DCD rROMCON1           ;
        DCD rROMCON2           ; 0x0400000 ~ 0x05FFFFF, ROM2
        DCD rROMCON3           ; 0x0600000 ~ 0x07FFFFF, ROM3
        DCD rROMCON4           ; 0x0800000 ~ 0x09FFFFF, ROM4
        DCD rROMCON5           ;
        DCD rSDRAMCON0         ; 0x1000000 ~ 0x13FFFFF, DRAM0 4M,
        DCD rSDRAMCON1         ; 0x1400000 ~ 0x17FFFFF, DRAM1 4M,
        DCD rSDRAMCON2         ; 0x1800000 ~ 0x1EFFFFF, DRAM2 16M
        DCD rSDRAMCON3         ; 0x1C00000 ~ 0x1FFFFFF
        DCD rSREFEXTCON        ; External I/O, Refresh


;=========================================
; Exception Handler Vector Table Entry Point
;=========================================
ExceptionHandlerTable
        DCD    UserCodeArea
        DCD    SystemUndefinedHandler
        DCD    SystemSwiHandler
        DCD    SystemPrefetchHandler
        DCD    SystemAbortHandler
        DCD    SystemReserv
        DCD    SystemIrqHandler
        DCD    SystemFiqHandler
       ALIGN
```

The above data table for Exception handler, System memory map will be defined at ROM or DRAM read only area. System initialization data table named SystemInitData and SystemInitDataSDRAM contain the register setting values for memory access cycle, refresh cycle, data bus width etc. You can update and change this parameters easily using EQU table . (**snds.a**)

Listing 3-11. (snds.a)

```
;/*************************************************************************/
;/* Format of the Program Status Register                                 */
;/*************************************************************************/
FBit            EQU    &40
IBit            EQU    &80
LOCKOUT         EQU    &C0      ;Interrupt lockout value
LOCK_MSK        EQU    &C0      ;Interrupt lockout mask value
MODE_MASK       EQU    &1F      ;Processor Mode Mask
UDF_MODE        EQU    &1B      ;Undefine Mode(UDF)
ABT_MODE        EQU    &17      ;Abort Mode(ABT)
SUP_MODE        EQU    &13      ;Supervisor Mode (SVC)
IRQ_MODE        EQU    &12      ;Interrupt Mode (IRQ)
FIQ_MODE        EQU    &11      ;Fast Interrupt Mode (FIQ)
USR_MODE        EQU    &10      ;User Mode(USR)

;/*************************************************************************/
;/*  SYSTEM STACK MEMORY  : 8K bytes system stacks are defined at memory.a
;/*************************************************************************/
USR_STACK_SIZE  EQU    1024
UDF_STACK_SIZE  EQU     512
ABT_STACK_SIZE  EQU     512
IRQ_STACK_SIZE  EQU    2048
FIQ_STACK_SIZE  EQU    2048
SUP_STACK_SIZE  EQU    2048

;/*************************************************************************/
;/*  SYSTEM CLOCK                                                         */
;/*************************************************************************/
MHz         EQU    1000000

;#ifdef KS32C50100
fMCLK_MHz   EQU    50000000   ; 50MHz, KS32C50100
;#else
;;fMCLK_MHz   EQU        33000000   ; 33MHz, KS32C5000
;fMCLK_MHz   EQU        40000000   ; 33MHz, KS32C5000
;#endif

fMCLK       EQU    fMCLK_MHz/MHz

;/*************************************************************************/
;/*  SYSTEM MEMORY CONTROL REGISTER EQU TABLES                            */
;/*************************************************************************/
;
;/* -> EXTDBWTH : Memory Bus Width register */
;-------------------------------------------------------------
;
```

```
DSR0            EQU  2:SHL:0       ; ROM0, 0 : Disable
                                   ;                          1 : Byte
                                   ;                          2 : Half-Word
                                   ;                          3 : Word
DSR1            EQU  2:SHL:2        ; ROM1
DSR2            EQU  3:SHL:4        ; ROM2
DSR3            EQU  3:SHL:6        ; ROM3
DSR4            EQU  3:SHL:8        ; ROM4
DSR5            EQU  3:SHL:10       ; ROM5
DSD0            EQU  3:SHL:12       ; DRAM0
DSD1            EQU  3:SHL:14       ; DRAM1
DSD2            EQU  3:SHL:16       ; DRAM2
DSD3            EQU  3:SHL:18       ; DRAM3
DSX0            EQU  3:SHL:20       ; EXTIO0
DSX1            EQU  3:SHL:22       ; EXTIO1
DSX2            EQU  3:SHL:24       ; EXTIO2
DSX3            EQU  3:SHL:26       ; EXTIO3

rEXTDBWTH   EQU
DSR0+DSR1+DSR2+DSR3+DSR4+DSR5+DSD0+DSD1+DSD2+DSD3+DSX0+DSX1+DSX2+DSX3
;-----------------------------------------------------------

;/* -> ROMCON0 : ROM Bank0 Control register */
;-----------------------------------------------------------
ROMBasePtr0     EQU  0x000:SHL:10   ;=0x0000000
ROMEndPtr0      EQU  0x020:SHL:20   ;=0x0200000
PMC0            EQU  0x0            ; 0x0=Normal ROM, 0x1=4Word Page
                                    ; 0x2=8Word Page, 0x3=16Word Page
rTpa0           EQU  (0x0:SHL:2)    ; 0x0=5Cycle, 0x1=2Cycle
                                    ; 0x2=3Cycle, 0x3=4Cycle
rTacc0          EQU  (0x6:SHL:4)    ; 0x0=Disable, 0x1=2Cycle
                                    ; 0x2=3Cycle, 0x3=4Cycle
                                    ; 0x4=5Cycle, 0x5=6Cycle
                                    ; 0x6=7Cycle, 0x7=Reserved
rROMCON0    EQU  ROMEndPtr0+ROMBasePtr0+rTacc0+rTpa0+PMC0
;-----------------------------------------------------------


;/* -> ROMCON1 : ROM Bank1 Control register */
;-----------------------------------------------------------
```

SOME CODE OMITTED FOR READIBILITY

```
rROMCON5    EQU  ROMEndPtr5+ROMBasePtr5+rTacc5+rTpa5+PMC5
;-----------------------------------------------------------

;/* -> DRAMCON0 : RAM Bank0 control register */
;-----------------------------------------------------------
EDO_Mode0       EQU  1                ;(EDO)0=Normal, 1=EDO DRAM
```

```
CasPrechargeTime0  EQU  0                  ;(Tcp)0=1cycle,1=2cycle
CasStrobeTime0     EQU  1                  ;(Tcs)0=1cycle ~ 3=4cycle
DRAMCON0Reserved   EQU  1                  ; Must be set to 1
RAS2CASDelay0      EQU  0                  ;(Trc)0=1cycle,1=2cycle
RASPrechargeTime0  EQU  2                  ;(Trp)0=1cycle ~ 3=4clcyle
DRAMBasePtr0       EQU  0x100:SHL:10       ;=0x1000000
DRAMEndPtr0        EQU  0x140:SHL:20       ;=0x1400000
NoColumnAddr0      EQU  2                  ;0=8bit,1=9bit,2=10bit,3=11bits
;-------------------------------------------------------------
Tcs0               EQU  CasStrobeTime0:SHL:1
Tcp0               EQU  CasPrechargeTime0:SHL:3
dummy0             EQU  DRAMCON0Reserved:SHL:4 ; dummy cycle
Trc0               EQU  RAS2CASDelay0:SHL:7
Trp0               EQU  RASPrechargeTime0:SHL:8
CAN0               EQU  NoColumnAddr0:SHL:30
;
rDRAMCON0   EQU  CAN0+DRAMEndPtr0+DRAMBasePtr0+Trp0+Trc0+Tcp0+Tcs0+dummy0+EDO_Mode0
;-------------------------------------------------------------------------------
SRAS2CASDelay0     EQU  1                  ;(Trc)0=1cycle,1=2cycle
SRASPrechargeTime0 EQU  3                  ;(Trp)0=1cycle ~ 3=4clcyle
SNoColumnAddr0     EQU  0                  ;0=8bit,1=9bit,2=10bit,3=11bits
SCAN0              EQU  SNoColumnAddr0:SHL:30
STrc0              EQU  SRAS2CASDelay0:SHL:7
STrp0              EQU  SRASPrechargeTime0:SHL:8
;
rSDRAMCON0     EQU  SCAN0+DRAMEndPtr0+DRAMBasePtr0+STrp0+STrc0
;-------------------------------------------------------------

 SOME CODE OMITTED FOR READIBILITY

;-------------------------------------------------------------
rSDRAMCON3                      EQU     SCAN3+DRAMEndPtr3+DRAMBasePtr3+STrp3+STrc3

;-------------------------------------------------------------

;/* -> REFEXTCON : External I/O & Memory Refresh cycle Control Register */
;-------------------------------------------------------------
RefCycle       EQU   16   ;Unit [us], 1k refresh 16ms
;RefCycle       EQU   8   ;Unit [us], 1k refresh 16ms
CASSetupTime    EQU   0    ;0=1cycle, 1=2cycle
CASHoldTime     EQU   0    ;0=1cycle, 1=2cycle, 2=3cycle,
                   ;3=4cycle, 4=5cycle,
RefCycleValue   EQU   ((2048+1-(RefCycle*fMCLK)):SHL:21)
Tcsr           EQU   (CASSetupTime:SHL:20)   ; 1cycle
Tcs            EQU   (CASHoldTime:SHL:17)
ExtIOBase       EQU   0x18360      ; Refresh enable, VSF=1
;
rREFEXTCON      EQU   RefCycleValue+Tcsr+Tcs+ExtIOBase
;-------------------------------------------------------------
;SRefCycle       EQU   16   ;Unit [us], 4k refresh 64ms
SRefCycle       EQU   8   ;Unit [us], 4k refresh 64ms
ROWcycleTime    EQU   3    ;0=1cycle, 1=2cycle, 2=3cycle,
                   ;3=4cycle, 4=5cycle,
```

```
SRefCycleValue  EQU   ((2048+1-(SRefCycle*fMCLK)):SHL:21)
STrc            EQU   (ROWcycleTime:SHL:17)
rSREFEXTCON     EQU   SRefCycleValue+STrc+ExtIOBase
;---------------------------------------------------------------

;
;/*************************************************************************/
;/* KS32C50100 SPECIAL REGISTERS                                          */
;/*************************************************************************/
;
ASIC_BASE      EQU     0x3ff0000

;/* Interrupt Control */

INT_CNTRL_BASE  EQU     ASIC_BASE+0x4000 ;Define base of all interrupt
                                         ; controller registers
IntMode                 EQU     ASIC_BASE+0x4000
IntPend         EQU     ASIC_BASE+0x4004
IntMask         EQU     ASIC_BASE+0x4008
INTOFFSET       EQU     ASIC_BASE+0x4024

; /* I/O Port Interface */
IOPMOD                  EQU     ASIC_BASE+0x5000
IOPCON          EQU     ASIC_BASE+0x5004
IOPDATA         EQU     ASIC_BASE+0x5008

;/*  UART 0,1  */
UARTLCON0    EQU     ASIC_BASE+0xD000
```

SOME CODE OMITTED FOR READIBILITY

```
; /* TIMER 0,1 */
TIMER_BASE  EQU     ASIC_BASE+0x6000 ;Define base for all timer
                                     ; registers
;/*************************************************************/
      END
```

Listing 3-12. (init.s)

```
        ;=====================================
        ; Initialise memory required by C code
        ;=====================================
        IMPORT  |Image$$RO$$Limit|  ; End of ROM code (=start of ROM data)
        IMPORT  |Image$$RW$$Base|   ; Base of RAM to initialise
        IMPORT  |Image$$ZI$$Base|   ; Base and limit of area
        IMPORT  |Image$$ZI$$Limit|  ; to zero initialise

        LDR  r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
        LDR  r1, =|Image$$RW$$Base|  ; and RAM copy
        LDR  r3, =|Image$$ZI$$Base|  ; Zero init base => top of initialised data
        CMP     r0, r1              ; Check that they are different
        BEQ     %1
0       CMP     r1, r3              ; Copy init data
        LDRCC   r2, [r0], #4
        STRCC   r2, [r1], #4
        BCC     %0
1       LDR     r1, =|Image$$ZI$$Limit| ; Top of zero init segment
        MOV     r2, #0
2       CMP     r3, r1              ; Zero init
        STRCC   r2, [r3], #4
        BCC     %2
```

Listing 3-13. (init.s)

```
          ;=================================================
        ; Now change to user mode and set up user mode stack.
        ;=================================================
      MRS     r0, cpsr
        BIC       r0, r0, #LOCKOUT | MODE_MASK
        ORR     r1, r0, #USR_MODE
        MSR     cpsr, r0
        LDR     sp, =USR_STACK

;       /* Call C_Entry application routine with a pointer to the first */
;       /* available memory address after ther compiler's global data   */
;       /* This memory may be used by the application.                  */
        ;===========================
        ; Now we enter the C Program
        ;===========================

      IMPORT  C_Entry
      BL      C_Entry
```

INTERRUPT HANDLING

Exception vector address and exception handling flows are previously described in this section. (**Table 3-2**, **Figure 3-4**).

Now, the interrupt handling method of NetMCU devices will be described hear. NetMCU devices, KS32C5000 / 5000A /50100, have a total 21 interrupt sources. Five special registers used to control the interrupt generation and handling. Five special registers are as follows: Interrupt mode register (INTMOD), Interrupt mask register (INTMSK), Interrupt pending register (INTPND), Interrupt offset register (INTOFFSET), Interrupt priority registers (INTPRI0,1,2,3,4,5).

Figure 3-4. show the ISR(Interrupt Service Routine) setup concept diagram. Interrupt handling C-source codes are listed in Listing 3-13.

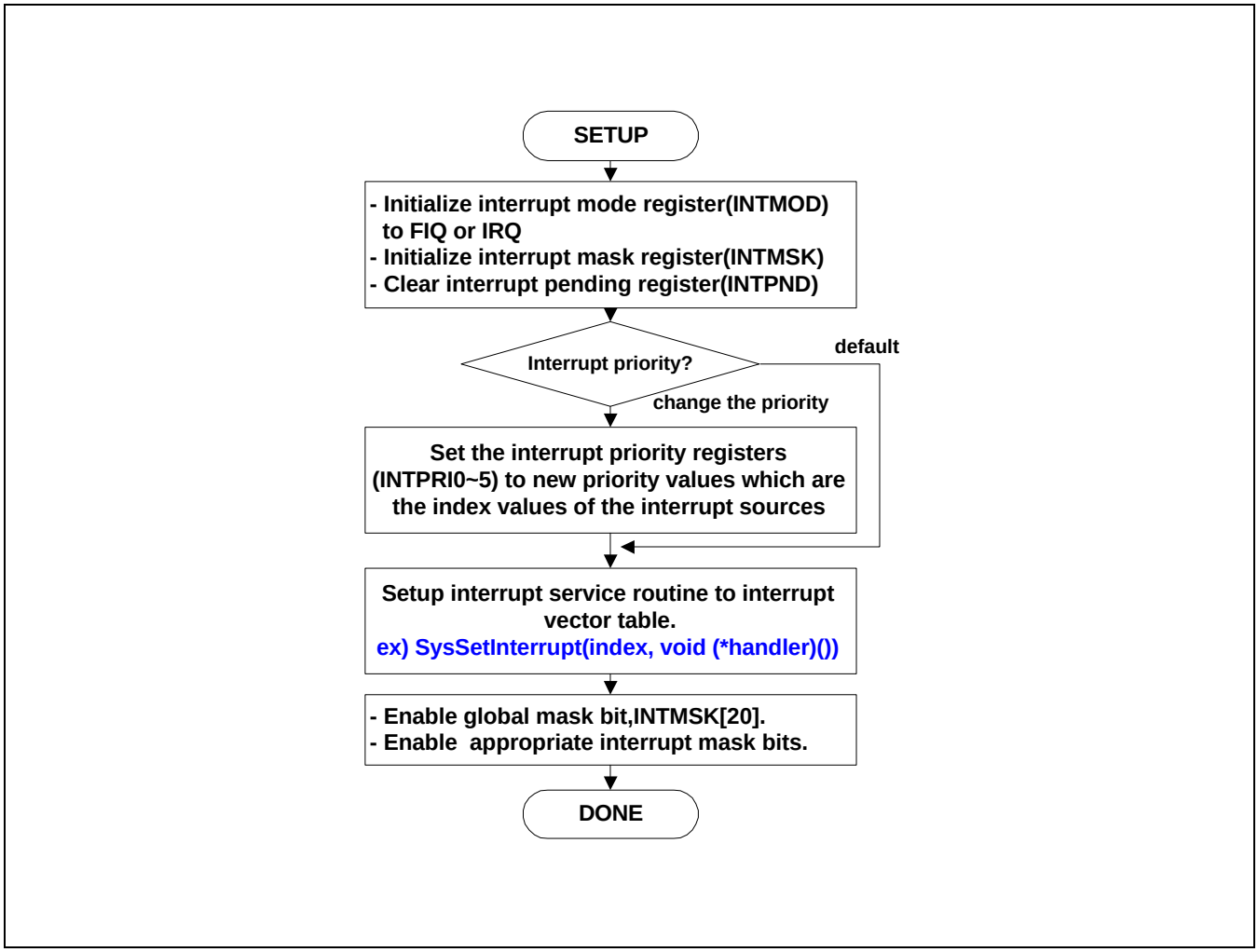


Figure 3-5. Setup concept diagrams for interrupt service routine

Listing 3-14. (isr.c)

```
void ClrIntStatus(void)
{
    INTMASK = 0x3fffff;  // All interrupt disabled include global bit
    INTPEND = 0x1fffff;  // All clear pending
    INTMODE = 0x1fffff;  // All FIQ mode

    /*
     * Interrupt Priority Reset value. Need to be changed priority,
     * Set interrupt priority register. In this case, Global interrupt
     * mask bit must be disabled.
     *
     */
   #ifdef  INT_PRIORITY
    INTPRI0 = 0x03020100;
    INTPRI1 = 0x07060504;
    INTPRI2 = 0x0b0a0908;
    INTPRI3 = 0x0f0e0d0c;
    INTPRI4 = 0x13121110;
    INTPRI5 = 0x00000014;
    INTOFFSET =  Read Only Register
   #endif
}


/*****************************************/
/****    Exception Handler Function    ****/
/*****************************************/
void      ISR_UndefHandler(REG32 *adr)
{
          //Print("\n** Trap : Undefined Handler\n") ;
          Print("\rUndefined Address : %08x ",adr);
          Print("\rUndefined Data : %08x ",*adr);
}

void      ISR_PrefetchHandler(REG32 *adr)
{
          //Print("\n** Trap : Prefetch Abort Handler\n") ;
          Print("\rPrefetch Abort Address : %08x ",adr);
          Print("\rPrefetch Abort Data : %08x ",*adr);
}

void      ISR_AbortHandler(REG32 *adr)
{
          //Print("\n** Trap : Data Abort Handler\n") ;
          Print("\rData Abort Address : %08x ",adr);
          Print("\rData Abort Data : %08x ",*adr);
}

void      ISR_SwiHandler(void)
{
```

```
        Print("\r** Trap : SWI Handler\n") ;
}


void    ISR_IrqHandler(void)
{
     IntOffSet = (U32)INTOFFSET;
     Clear_PendingBit(IntOffSet>>2) ;
    (*InterruptHandlers[IntOffSet>>2])(); // Call interrupt service routine

}


void    ISR_FiqHandler(void)
{
    IntOffSet = (U32)INTOFFSET;
    Clear_PendingBit(IntOffSet>>2) ;
    (*InterruptHandlers[IntOffSet>>2])(); // Call interrupt service routine
}


/*************************************************************************/
/* InitIntHandlerTable: Initialize the interrupt handler table       */
/*     NOTE(S): This should be called during system initialization    */
/*************************************************************************/
void InitIntHandlerTable(void)
{
        REG32 i;

        for (i = 0; i < MAXHNDLRS; i++)
                InterruptHandlers[i] = DummyIsr;
}

/*********************************************************/
/* SysSetInterrupt: Setup Interrupt Handler Vector Table */
/*********************************************************/
void SysSetInterrupt(REG32 vector, void (*handler)())
{
        InterruptHandlers[vector] = handler;

}

/*********************************************************/
/* InitInterrupt: Initialize Interrupt             */
/*********************************************************/
void InitInterrupt(void)
{
    ClrIntStatus();     // Clear All interrupt
        InitIntHandlerTable() ;
}
```

The interrupt offset register [INTOFFSET] is a special function of the NetMCU device that solves the interrupt latency problem caused by many interrupt resources. INTOFFSET has the index value of the top priority interrupt which already is enabled by the interrupt mask bit[INTMSK] and is pending on the interrupt pending register [INTPND]. Therefore, the appropriate interrupt service routine in interrupt vector table which is defined as function array can be serviced by reference the value of INTOFFSET.

For examples,

```
void      ISR_IrqHandler(void)
{
      IntOffSet = (U32)INTOFFSET;
      Clear_PendingBit(IntOffSet>>2) ;
     (*InterruptHandlers[IntOffSet>>2])(); // Call interrupt service routine

}
```

Interrupt pending bit also cleared in this interrupt handler.

An example of the interrupt handling process

Figure 3-6 shows the IRQ interrupt handling process. For the more easy understand about interrupt handling process, I am going to explain the IRQ interrupt handling process as an example.

Upon the ARM7 Series architecture, the exception vector table is on the address 0x0 to 0x1C. So Boot ROM have to be located at address zero. After the power on reset or reset signal input, boot ROM exception vector table is mirrored to DRAM exception vector tables using default exception handler.

For example, the IRQ interrupt vector address,0x18, has the address of IRQ_Handler. Using this default handler, IRQ vector table mirrored to DRAM IRQ handler vector table. The IRQ handler vector table on DRAM will be initialized by SystemIrqHandler during the boot processing. SystemIrqHandler will call the C-coded IRQ handler, ISR_IrqHandler() when the IRQ interrupt occurred.

DRAM memory map is difined at **memory.a** (Listing 3-13)**.**

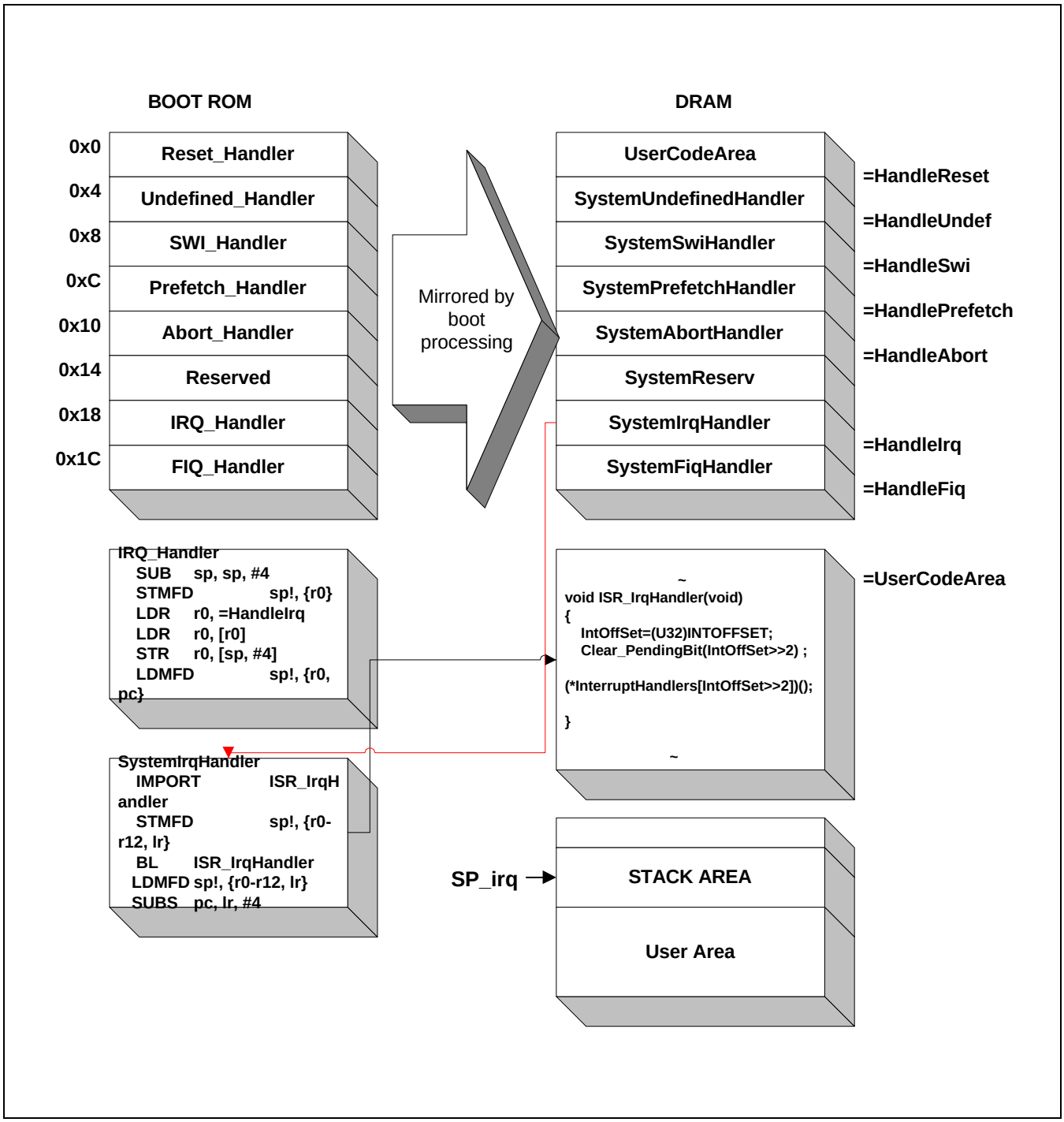


Figure 3-6. Example of IRQ interrupt handling process

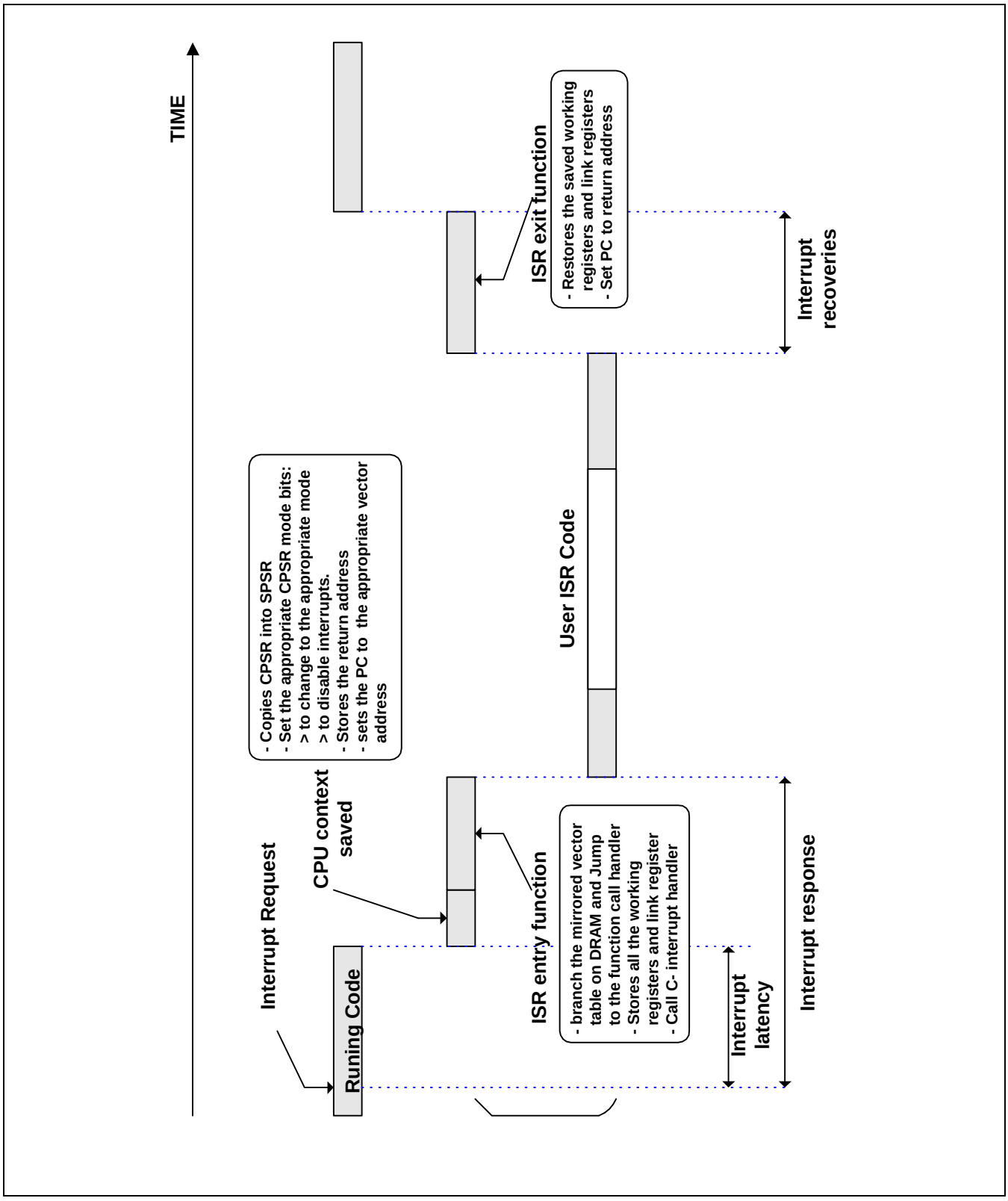


Figure 3-7. Interrupt Latency, Response and Recoveries of NetMCU Device

Interrupt latency, interrupt response and recoveries

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchronizer(*Tsyncmax* if asynchronous), plus the time for the longest instruction to complete (*Tldm*,the longest instruction is an LDM which loads all the registers including PC), plus the time for the data abort entry (*Texe*), plus the time for FIQ entry (*Tfig*). At the end of this time ARM7TDMI will be executing the instruction at 0x1C.

Tsyncmax is 3 processor cycles, *Tldm* is 20 cycles, *Texe* is 3 cycles, and *Tfiq* is 2 cycles. The total time is therefore 28 processor cycles.

Interrupt Latency = Tsyncmax (3cycles) + Tldm (20cycles) + Texe (3cycles) + Tfiq (2cycles) = 28 cycles.

The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time. The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchronizer(*Tsyncmin*), plus *Tfiq*. This is 4 processor cycles.

INSTRUCTION & DATA CACHE / INTERNAL SRAM

The NetMCU devices has the unified(Instruction / Data) cache. This cache memory can be configured to internal SRAM using system configuration register[SYSCFG].

Listing 3-15, the C source code demonstrate the usage of Cache memory. It can be configured as 4K cache/4K SRAM or all cache or SRAM depending on the set values of the SYSCFG register.

Listing 3-15. (system.c)

```
void SyscfgInit(int cm)
{
    /* Disable cache before cache mode change */
    SYSCFG &= ~(STALL|CACHE|CACHE_MODE|WRITE_BUFF);

    switch(cm) {
        case  0 : /* 4K Cache,4K SRAM */
                SYSCFG |=(CACHE_MODE_00|WRITE_BUFF);
                CacheFlush();
                SYSCFG |=CACHE; /* Cache Enable */
                break;

        case  1 : /* 8K Cache */
                SYSCFG |=(CACHE_MODE_01|WRITE_BUFF);
                CacheFlush();
                SYSCFG |=CACHE; /* Cache Enable */
                break;

        case  2 : /* CACHE OFF: 8K SRAM */
                SYSCFG |=(CACHE_MODE_10|WRITE_BUFF);
                CacheFlush();

        default : break;
    }
```

Cache Flush Operation

If you change the cache mode or memory map configurations when the cache is enabled, then memory consistency problem will be occurred. So, in advance that, you have to clear the TAG RAM for flush the cache.

Listing 3-16. (system.c)

```
/*
 *  Cache flush function for re-configuration cache mode
 *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 */
void CacheFlush(void)
{
    int i;
    unsigned int *tagram;

    tagram = (unsigned int *)TagRAM;
    SYSCFG &= ~CACHE;  // Disable cache before cache flush

    for(i=0; i < 256; i++) {
        *tagram = 0x00000000;   // Clear tag ram
        tagram += 1;
    }

    //Print("\nCache Flushed!!\r");
}
```

IIC BUS CONTROLLER

The NetMCU device' s 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 NetMCU device' s IIC interface.

For the purpose of more understanding about IIC serial interface protocol, I am about to introduce the function description of the KS24L321/641. 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

I²C-Bus Interface

The KS24L321/641 supports the I²C-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 I²C-bus as slaves (see Figure 3-8). 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

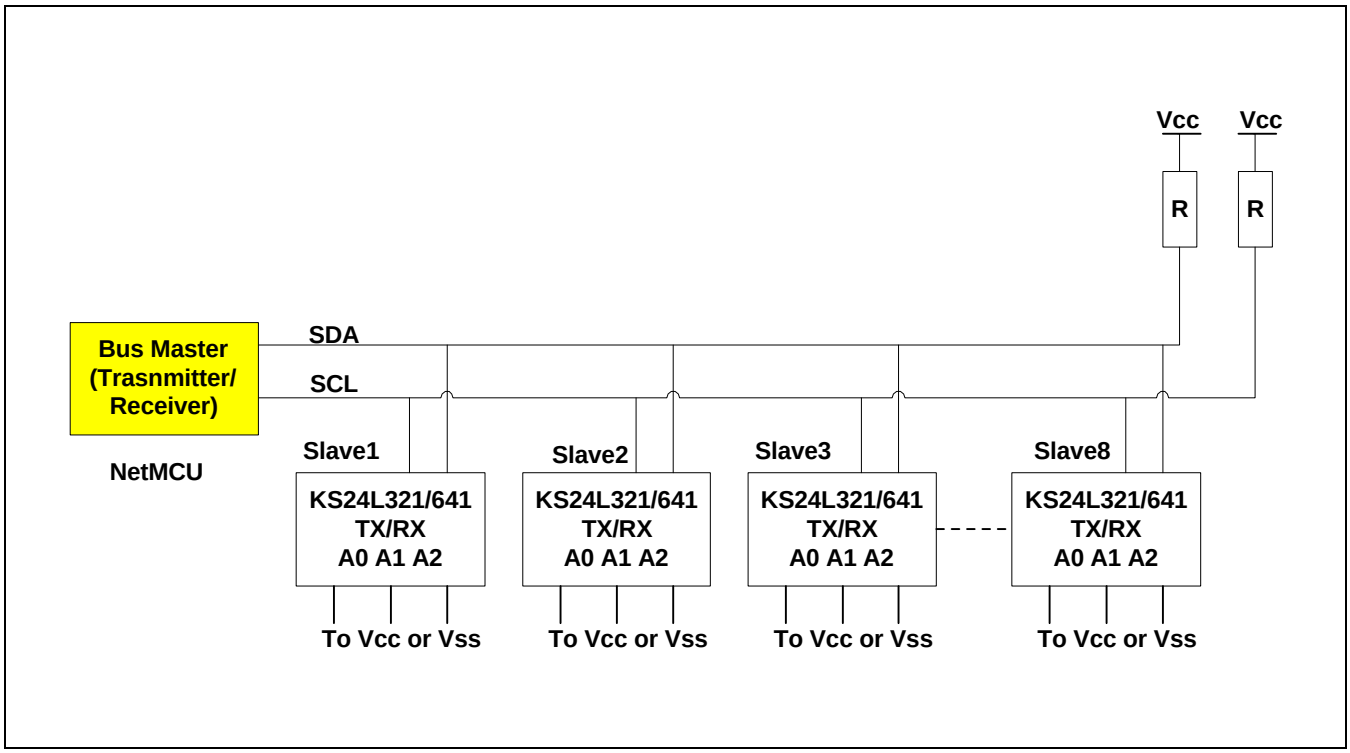


Figure 3-8. Typical Configuration

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 3-9).

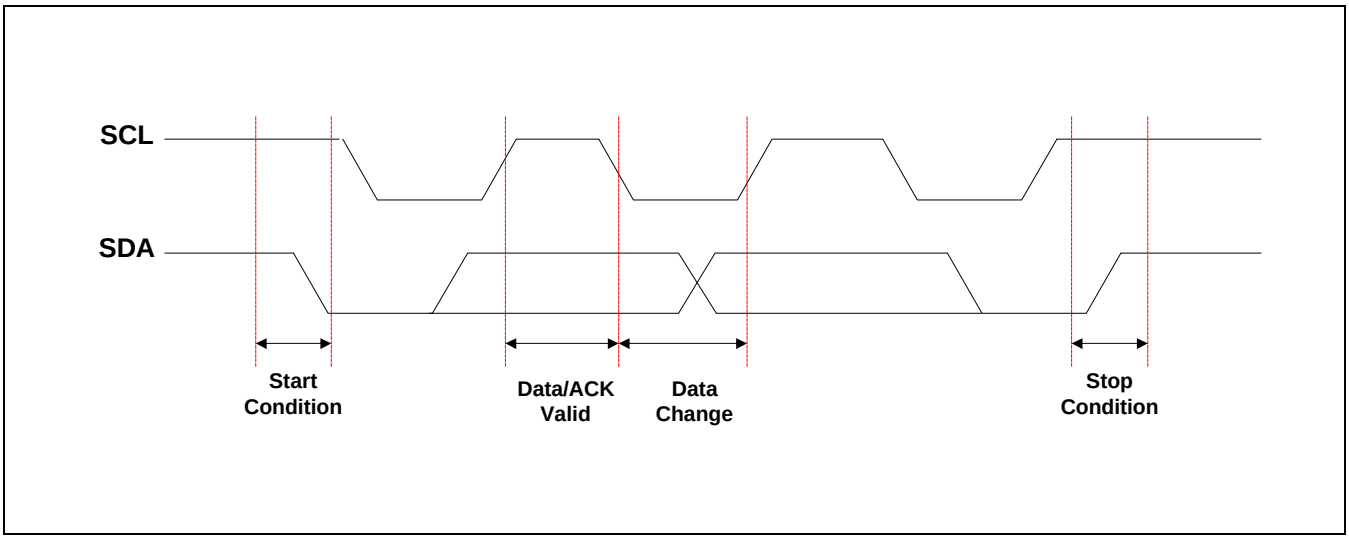


Figure 3-9. Data Transmission Sequence

— **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 3-10). 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.

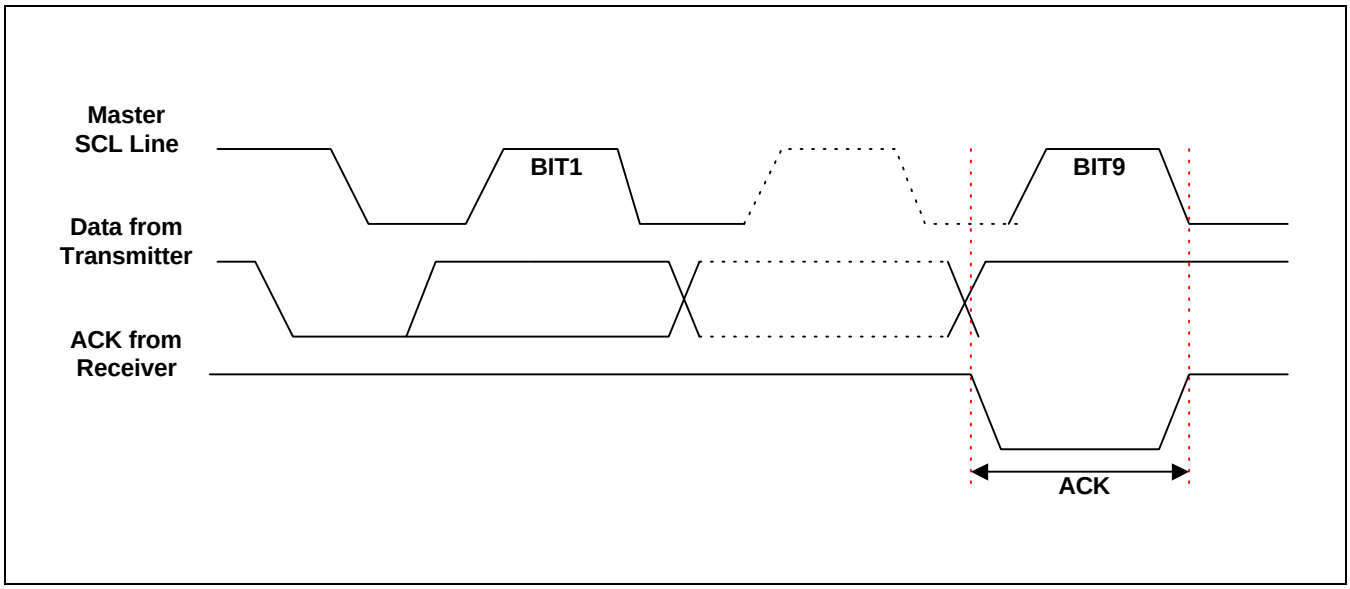


Figure 3-10. Acknowledge Response From Receiver

— **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 (see Figure 3-11 below).

— **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

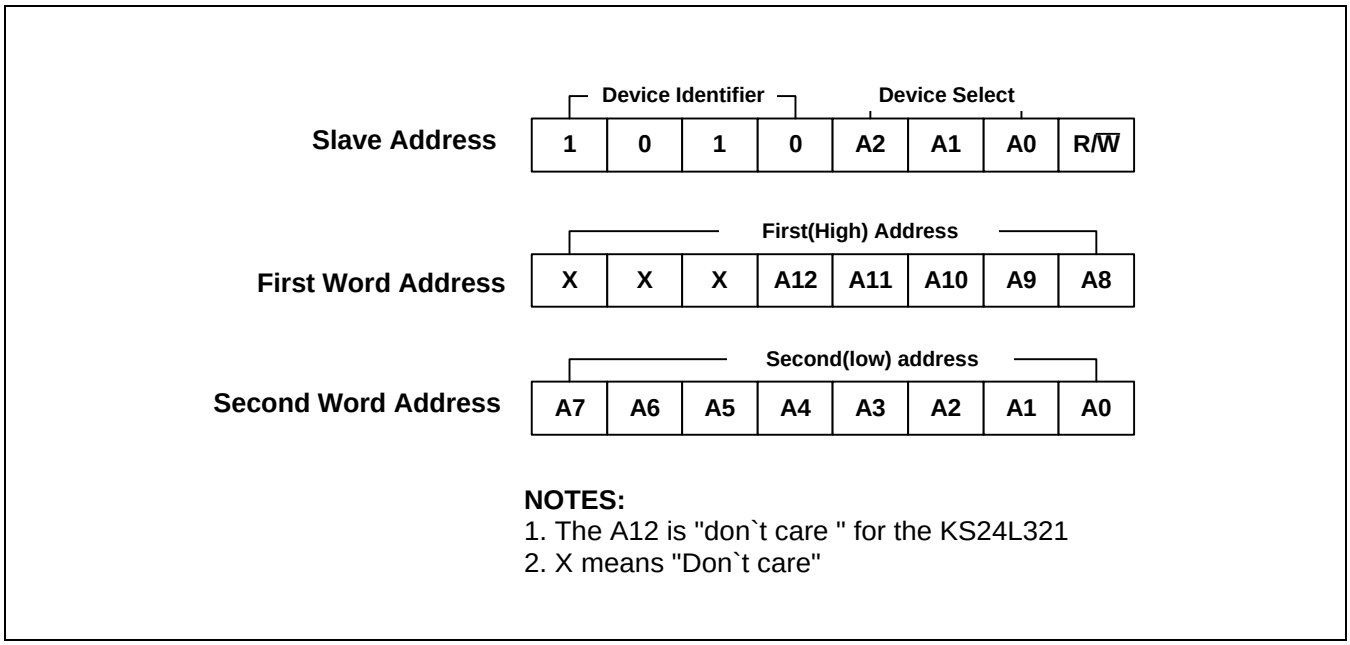


Figure 3-11. Device Address

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.

Listing 3-17 shows the IIC initialize C-routine. Where, the IIC prescaler frequency,fSCL is defined at sysconf.h which is the SNDS100 target system configuration header file.

Listing 3-17. (iic.c)

```
/*************************************************************
 *                                                           *
 *                IIC SETUP ROUTINE                          *
 *                                                           *
 *************************************************************/
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
}

/*************************************************************
 *                                                           *
 *    SETUP IIC PRESCALER VALUE FROM SERIAL CLOCK FREQUENCY      *
 *                                                           *
 *************************************************************/
int  SetPreScaler(int sclk)
{
    return((int)(((fMCLK/sclk)-3.0)/16.0)-0.5);  //add 0.5 for
}
```

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 3-12).

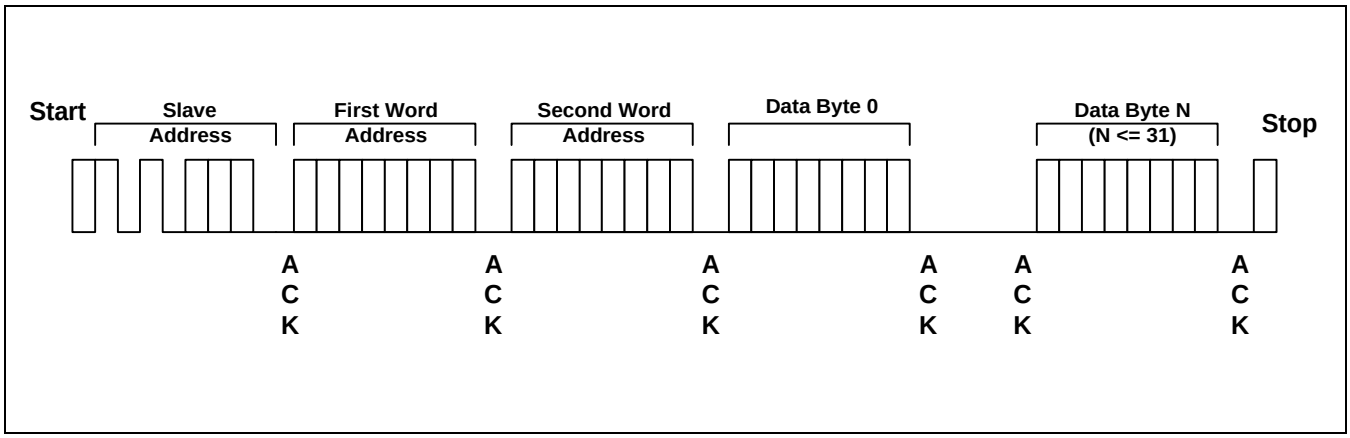


Figure 3-12. Page Write Operation

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(Listing 3-18). This write function divide the transfer data to page size and transmit it to Slave until all transfer data is sent. NetMCU 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 difined at IIC header file(**iic.h**).

Listing 3-18. (iic.c)

```
/*****************************************************************
 *                                                               *
 *         LIBRARY FUNCTIONS FOR IIC READ & WRITE                *
 *           KS24C32/64   : IIC Serial EEPROM                    *
 *                                                               *
 *****************************************************************/
void IICWriteInt(U8 SlaveAddr,U32 WriteAddr,U8 *data,U32 SizeOfData)
{
    int page,j;
    int no_of_page;      /* Number of page */
    int remain_byte;
    U32 PageAccessAddr;


    SysSetInterrupt(nIIC_INT,IICWriteIsr) ; /*Setup IIC Tx interrupt */
    Enable_Int(nIIC_INT) ;

    PageAccessAddr = WriteAddr;
    iic_txmit.SLAVE_ADDR = SlaveAddr;

    no_of_page = (int)(ceil(SizeOfData/(U32)SizeOfPage));
    remain_byte = (int)(SizeOfData%(U32)SizeOfPage);

    for(page=0; page <= no_of_page;page++)
    {
        if(SizeOfData < SizeOfPage)
          for(j=0; j < SizeOfData; j++)
              iic_txmit.PAGE_BUFFER[j] = *data++;
          iic_txmit.WriteDataSize = SizeOfData;
        }
        else {
              if(page == no_of_page) {
                 for(j=0; j < remain_byte; j++)
                     iic_txmit.PAGE_BUFFER[j] = *data++;
                 iic_txmit.WriteDataSize = remain_byte;
              }
              else {
                  for(j=0; j < SizeOfPage; j++)
                      iic_txmit.PAGE_BUFFER[j] = *data++;
                  iic_txmit.WriteDataSize = SizeOfPage;
              }
        }

        IicSetup();
        iic_txmit.FLAG = 0x0;
        iic_txmit.BuffByteCnt = 0x0;
        iic_txmit.BYTE_ADDR_MSB = (U8)((PageAccessAddr>>8) & 0xff);
        iic_txmit.BYTE_ADDR_LSB = (U8)(PageAccessAddr & 0xff);
```

```
        /* Step 1: Setup IICON register for transmit start */
        while(IICCON & BUSY);   /* Wait! the iic bus is busy */
        IICCON = START|ACK|IEN; /* Now, Start to transmit */

        /* Send Slave Address and Write command */
        IICBUF = iic_txmit.SLAVE_ADDR|S_WRITE;

        while(!(iic_txmit.FLAG & iic_page_tx_done));
        PageAccessAddr += SizeOfPage;
        for(j=0; j< (int)Write_Cycle_ms(5); j++);  /* for 5ms write cycle */
    }
}



/*****************************************************************
 *                                                               *
 *         IIC INTERRUPT SERVICE ROUTINES                        *
 *                                                               *
 *****************************************************************/
void IICWriteIsr(void)
{
    if(!(iic_txmit.FLAG & (U32)iic_byte_addr_msb))
        /* Send byte address: MSB */
        IICBUF = iic_txmit.BYTE_ADDR_MSB;
        iic_txmit.FLAG |= (U32)iic_byte_addr_msb;
    }
    else if(!(iic_txmit.FLAG & (U32)iic_byte_addr_lsb)) {
        /* Send byte address: LSB */
        IICBUF = iic_txmit.BYTE_ADDR_LSB;
        iic_txmit.FLAG |= (U32)iic_byte_addr_lsb;
    }
    else if(iic_txmit.BuffByteCnt < iic_txmit.WriteDataSize) {
        IICBUF = iic_txmit.PAGE_BUFFER[iic_txmit.BuffByteCnt++];
    }
    else
          /* STOP IIC Controller */
          IICCON = STOP;
          /* byte data or page data transmit done */
          iic_txmit.FLAG |= (U32)iic_page_tx_done;
    }
}
```

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 (see Figure 3-13).

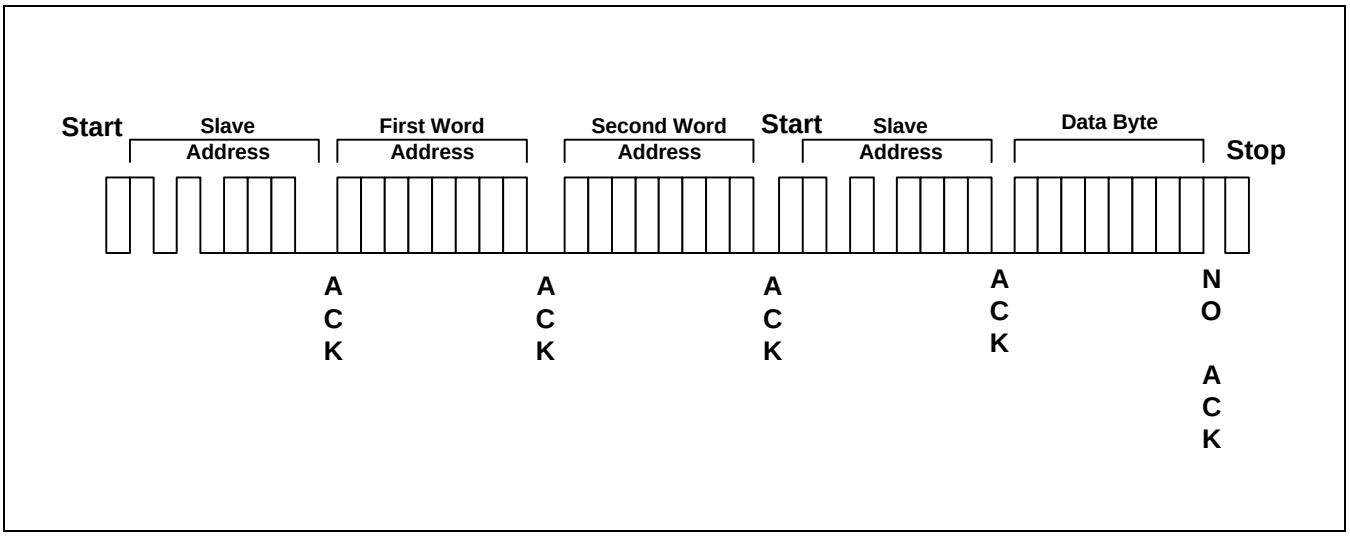


Figure 3-13. Random Address Byte Read Operation

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 (see Figure 3-14).

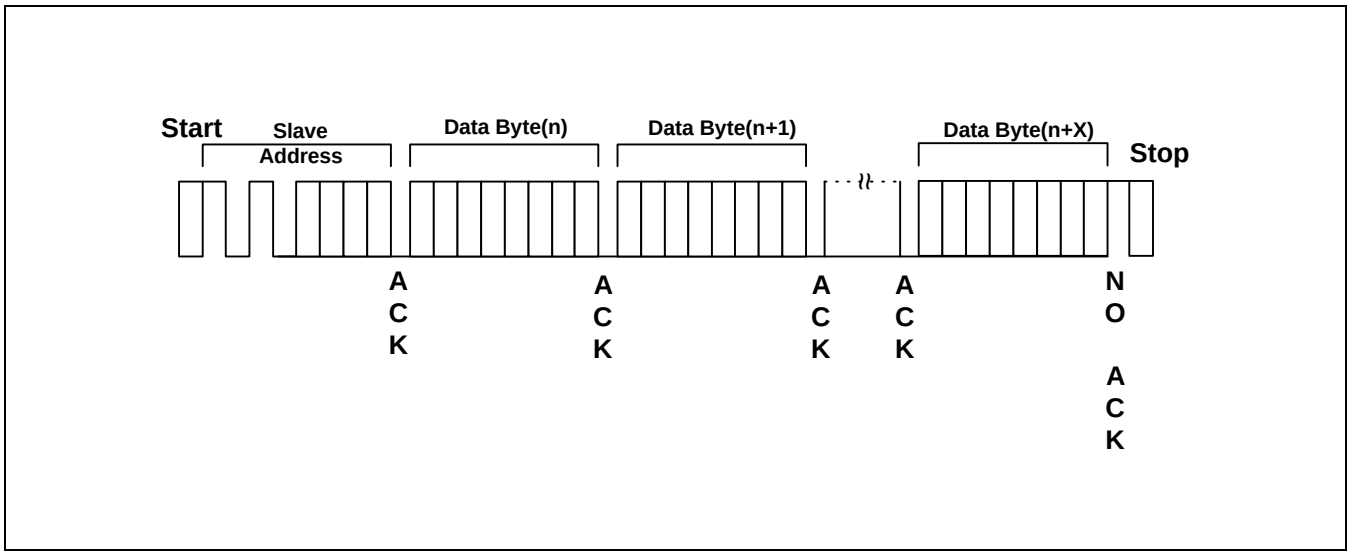


Figure 3-14. Sequential Read Operation

Listing 3-19. (iic.c)

```
void *IICReadInt(U8 SlaveAddr,U32 ReadAddr,U32 SizeOfData)
{
    U8 *ReadPtr;  /* data read pointer */

    IicSetup();
    SysSetInterrupt(nIIC_INT,IICReadIsr) ; /*Setup IIC Tx interrupt */
    Enable_Int(nIIC_INT) ;

    /*Memory alloc for receive data */
    if((ReadPtr = (U8 *)malloc((unsigned)SizeOfData)) == (U8 *)(NULL))
        Print("\rMemory allocation error occurred!!!\r");
    iic_recv.RCV_BUFFER = ReadPtr;

    iic_recv.FLAG = 0x0;
    iic_recv.ByteReadCnt = 0x0;
    iic_recv.ReadDataSize = SizeOfData;
    iic_recv.SLAVE_ADDR = SlaveAddr;
    iic_recv.BYTE_ADDR_MSB = (U8)((ReadAddr>>8) & 0xff);
    iic_recv.BYTE_ADDR_LSB = (U8)(ReadAddr & 0xff);

    /* Step 1: Setup IICON register for receive start */
    while(IICCON & BUSY);   /* Wait! the iic bus is busy */
    IICCON = START|ACK|IEN;

    /* Send Slave Address and Write command */
    IICBUF = iic_recv.SLAVE_ADDR|S_WRITE;

    while(!(iic_recv.FLAG & iic_byte_rx_done));
    return(ReadPtr);        /* return receive data pointer */
}

void IICReadIsr(void)
{
    if(!(iic_recv.FLAG & (U32)iic_byte_addr_msb))
    //else if(!(iic_recv.FLAG & (U32)iic_byte_addr_msb))
        /* Send byte address: MSB */
        IICBUF = iic_recv.BYTE_ADDR_MSB;
        iic_recv.FLAG |= (U32)iic_byte_addr_msb; /* send msb byte addr */
    }
    else if(!(iic_recv.FLAG & (U32)iic_byte_addr_lsb)) {
        /* Send byte address: LSB */
        IICBUF = iic_recv.BYTE_ADDR_LSB;
        iic_recv.FLAG |= (U32)iic_byte_addr_lsb; /* send lsb byte addr */
    }
    else if(!(iic_recv.FLAG & (U32)iic_repeat_start)) {
        /* Repeat Start */
        IICCON = RESTART;
        IICCON = START|ACK|IEN;
        IICBUF = iic_recv.SLAVE_ADDR|S_READ;
        iic_recv.FLAG |= (U32)iic_repeat_start;
```

```
    }
    else if(!(iic_recv.FLAG & (U32)iic_multi_recv)) {
        /* Receive multiple data */
        IICCON = ACK|IEN;
        iic_recv.FLAG |= (U32)iic_multi_recv;
    }
    else if(iic_recv.ByteReadCnt < iic_recv.ReadDataSize-1) {
        *(iic_recv.RCV_BUFFER)++ = IICBUF;
        iic_recv.ByteReadCnt++;
    }
    else if(!(iic_recv.FLAG & (U32)iic_no_more_recv))
          /* Now,no more received data is required from slave */
          IICCON = NOACK|IEN;
          iic_recv.FLAG |= (U32)iic_no_more_recv;
    }
    else { /* Receive last data and STOP */
          *(iic_recv.RCV_BUFFER)++ = IICBUF;

          /* STOP IIC Controller */
          IICCON = STOP;

          /* byte data receive done */
          iic_recv.FLAG |= (U32)iic_byte_rx_done;
    }
}
```

UART (UNIVERSAL ASYNCRONOUS RECEIVER/TRANSMITTER)

The KS32C5000(A)/50100 UART unit provides two channel serial communication ports. This UART unit were implemented and embedded as console port. So, it did not support full modem interface pins. It only support that four serial input or output pins which are the data transmit and receive pins(UATxD/UARxD) ,data set ready (nUADSR), data terminal ready(nUADTR) pin per channel.

If you needs the full modem interface pins, you can implement it use by general I/O ports or you have to use an commercial UART chip which will be described on the next section in this application notes.

The KS32C5000(A) UART features almost same as the KS32C50100 except it have 16bytes receive FIFO and one more interrupt source for receive error. (See User' s Manual for more details). But the listed UART driver sources can be used for the KS32C5000(A) and KS32C50100 commonly.

UART Baud Rate

You can choose the external clock or internal system clock as UART clock source to generate an appropriate baud rate. For the external UART clock, 29.4912MHz Oscillator is mounted on SNDS100 rev1.0 target board. If you are using KS32C5000 device, you have to choose internal system clock for UART because it did not support external UART input pin.

Program tips for buad rate calculation (buad50100.zip for KS32C50100, buad5k.zip for KS32C5000(A)) are available at our web site. The following baud rate tables are obtained from this program tips.

Table 3-2. Baud Rate Table for KS32C50100 (fMCLK = 25MHz)

UBRDIVn	TYPICAL[bps]	SPEC.[bps]	ERROR RATE[%]
0x5150	1200.1	1200	0.006
0x28a0	2400.2	2400	0.006
0x1450	4792.9	4800	-0.147
0x0a20	9585.9	9600	-0.147
0x0500	19290.1	19200	0.469
0x0280	38109.8	38400	-0.756
0x01a0	57870.4	57600	0.469
0x00d0	111607.1	115200	-3.119
0x0060	223214.3	230400	-3.119

Table 3-3. Baud Rate Table for KS32C5000(A) (fMCLK = 33MHz)

UBRDIVn	TYPICAL[bps]	SPEC.[bps]	ERROR RATE[%]
0x000d6	9593.0	9600	-0.073
0x0006b	19097.2	19200	-0.535
0x0006a	19275.7	19200	0.394
0x00035	38194.4	38400	-0.535
0x00023	57291.7	57600	-0.535
0x00011	114583.3	115200	-0.535
0x00008	229166.7	230400	-0.535

Table 3-4. Baud Rate Table for KS32C5000(A)/50100 (fMCLK = 29.4912MHz)

UBRDIVn	TYPICAL[bps]	SPEC.[bps]	ERROR RATE[%]
0x0bf0	9600.0	9600	0.000
0x05f0	19200.0	19200	0.000
0x02f0	38400.0	38400	0.000
0x01f0	57600.0	57600	0.000
0x00f0	115200.0	115200	0.000
0x0070	230400.0	230400	0.000
0x0030	460800.0	460860	0.013
0x0010	921600.0	921600	0.000

To initialize the UART baud rate divisor register[UBRDIVn], The following data structure used to reference the UBRDIVn register initialize value according to be given baud rate.

```
typedef struct
{
    uint32 baud;   /* baud rate: ex) 115200bps */
    uint32 div;    /* divisor register value for given baud rate */
}BaudTable;
```

You can get the divisor register setting value from BaudRateVal(uint32 baud) function. This function return the index value of baud rate table for be given UART baud rate. For example:

rBRDIVn = U_BaudRate[BaudRateVal(115200)].div;

If you select the external UART clock(29.4912MHz), the baud rate divisor value, 0x0000f, can be get after the above statement is run.

Listing 3-20. (uart.c)

```
BaudTable U_BaudRate[BAUD_TABLE] = {

#ifdef EX_UCLK
/* for 29.4912MHz UART clock */
         9600,  0x000bf,
        19200,  0x0005f,
        38400,  0x0002f,
        57600,  0x0001f,
        115200, 0x0000f,
        230400, 0x00007,
        460800, 0x00003
#else
#ifdef KS32C50100
/* for 50MHz/2 UART clock */
         9600,  0x00a20,
        19200,  0x00500,
        38400,  0x00280,
        57600,  0x001a0,
        115200, 0x000d0,
        230400, 0x00060,
        460800, 0x00020   // not available

#else
/* for 33MHz UART clock */
         9600,  0x000d6,
        19200,  0x0006a,
        38400,  0x00035,
        57600,  0x00023,
        115200, 0x00011,
        230400, 0x00008,
        460800, 0x00008   // not available
#endif
#endif
};


uint32 BaudRateVal(uint32 baud)
{
    uint32 index;

    for(index = 0; index < BAUD_TABLE; index++)
    {
        if(U_BaudRate[index].baud == baud) return(index);
    }

    return(0); /* baudrate data doesn't in table */
}
```

UART Initializations

All data structures for UART configuration and initialization are defined at UART C-header file.(**uart.h**)

Listing 3-20. (uart.c)

```
uint32  UART_Initialize()
{

    Disable_Int(nGLOBAL_INT);  /* Global interrupt disabled */

    /*****************************/
    /* Initialize UART channel 0 */
    /*****************************/
    uart_dev_init.com_port  = SERIAL_DEV0;    /* com 0 */
    uart_dev_init.baud_rate = baudrate;
    uart_dev_init.data_mode = (UCON_RXM_INTREQ|UCON_TXM_INTREQ|UCON_RXSTAT_INT);
    uart_dev_init.parity    = ULCON_PMD_NO;   /* No parity */
    uart_dev_init.stop_bits = 0;              /* one bit */
    uart_dev_init.data_bits = ULCON_WL8;      /* 8bits */
#ifdef EX_UCLK
    uart_dev_init.clk_sel = ULCON_UCLK;       /* external clock,29.4912MHz */
#else
    uart_dev_init.clk_sel = 0;                /* internal clock */
#endif
    UART_Init(&uart_dev_init);

    /*****************************/
    /* Initialize UART channel 1 */
    /*****************************/
    uart_dev_init.com_port = SERIAL_DEV1;     /* com 0 */
    UART_Init(&uart_dev_init);

    Enable_Int(nGLOBAL_INT);  /* Global interrupt disabled */

    return(SUCCESS);
}

/***************************/
/* Uart main Init function */
/***************************/
uint32 UART_Init(SERIAL_DEV  *s)
{
     uint32    rUARTBRD;

     /* UART interrupt off */
     UARTRxIntOff(s->com_port);
     UARTTxIntOff(s->com_port);

     /* Initialize UART transmit & receive Queue */
     TxQInit(s->com_port);
     RxQInit(s->com_port);
```

```
    /* default baud rate will be set. sysconf.h */
    rUARTBRD = U_BaudRate[BaudRateVal(s->baud_rate)].div;

    if(s->com_port)
    {
        /* Interrupt service routine setup */
       SysSetInterrupt(nUART1_TX_INT, Uart1TxLisr);
#ifdef KS32C50100
       SysSetInterrupt(nUART1_RX_ERR_INT, Uart1RxErrLisr);
#else
       SysSetInterrupt(nUART1_RX_INT, Uart1RxErrLisr);
       SysSetInterrupt(nUART1_ERROR_INT, Uart1RxErrLisr);
#endif

       UARTLCON1 = s->data_bits|s->stop_bits|s->parity|s->clk_sel;
       UARTCONT1 = s->data_mode;
       UARTBRD1  = rUARTBRD;
    }
    else
    {
       /* Interrupt service routine setup */
       SysSetInterrupt(nUART0_TX_INT, Uart0TxLisr);
#ifdef KS32C50100
       SysSetInterrupt(nUART0_RX_ERR_INT, Uart0RxErrLisr);
#else
       SysSetInterrupt(nUART0_RX_INT, Uart0RxErrLisr);
       SysSetInterrupt(nUART0_ERROR_INT, Uart0RxErrLisr);
#endif
        /* UART mode, default baud rate setup */
       UARTLCON0 = s->data_bits|s->stop_bits|s->parity|s->clk_sel;
       UARTCONT0 = s->data_mode;
       UARTBRD0  = rUARTBRD;
    }

    //UARTRxIntOn(s->com_port);
    //UARTTxIntOn(s->com_port);

    return(SUCCESS);
}
```

UART Polled I/O Functions

Put_char() and get_char() is implemented for to transmit and receive data through UART channel in polling method.

These poll function check the UART buffer status using by UART status register[USTATn] for data transfer. If the UART transmit holding register[UARTTXHn] is in empty state, then the byte data will be written to it. UARTTXHn

Register' s data will be shifted out from transmit shift register to UART transmit output pin[UATxDn] serially.

If the UART receive buffer register[URXBUFn] is filled with received data, the get_char() will return the byte data of UEXBUFn.

The formatted output polled I/O function, Print() is defined at pollio.c. This function is similar to printf(), ANSI C standard library function.

Listing 3-21. (uart.c)

```
void put_char(uint32 channel,char ch)
{
    if(channel) {
        WaitXmitter(UARTSTAT1);
        UARTTXH1 = ch;
    }
    else {
        WaitXmitter(UARTSTAT0);
        UARTTXH0 = ch;
    }
}

char get_char(uint32 channel)
{
  char ch;

    if(channel) { WaitRcver(UARTSTAT1);
                ch = UARTRXB1;
    }
    else { WaitRcver(UARTSTAT0);
          ch = UARTRXB0;
    }
    return ch;
}

void put_string(char *ptr )
{
    while(*ptr )
    {
        put_byte(*ptr++ );
    }
}
```

The Circular Queue

To transfer the UART data by interrupt method, the circular queue used for UART data structure as following:

```
/* Transmit & Receive Que data structure */
#define MAXEVENT 10    /* 10 bytes buffer */

typedef struct
{
   char  buff[MAXEVENT];      /* data buffer */
   int         wptr;          /* write pointer */
   int         rptr;          /* read pointer */
} UART_BUFFER;
```

To create a circular queue for use in UART I/O functions based on the interrupt method, the function TxQWr() for Transmit data write Queue and RxQRd() for receive data read queue and UART interrupt service routines for transmit/receive data from Queue.(Listing 3-22)

The UART interrupt service routines for data transferring are already registered at interrupt vector table at UART initialize function ,UART_Initialize(), is called. (Listing 3-20). If the UART Tx/Rx interrupt were enabled and an event occurred for Rx or Tx, then the UART interrupt service routines,Uart0TxLisr()/Uart0RxLisr() for channel 0, will be served.

Listing 3-22. (uart.c)

```
/* Transmit Que write function */
uint32 TxQWr(uint32 channel,uint8 data)
{
       if(TxQ[channel].wptr+1 == TxQ[channel].rptr)
       {
          return(ERROR); /* ring buffer full state */
       }

       TxQ[channel].buff[TxQ[channel].wptr++] = data;

       if(TxQ[channel].wptr == MAXEVENT)
       {
          TxQ[channel].wptr=0;
       }
       return(SUCCESS);
}


/* Receive Que read function */
uint8 RxQRd(uint32 channel)
{
    if(RxQ[channel].rptr == MAXEVENT)
         RxQ[channel].rptr=0; /*loop back*/

    if(RxQ[channel].rptr == RxQ[channel].wptr)
         return('\0');
```

```
    return(RxQ[channel].buff[RxQ[channel].rptr++]);
}


void Uart0TxLisr(void)
{

   if(UARTSTAT0 & USTAT_TXB_EMPTY)
   {
        if(TxQ[0].rptr == MAXEVENT)
             TxQ[0].rptr=0; /*loop back*/

        if(TxQ[0].rptr != TxQ[0].wptr)
        {
          UARTTXH0 = TxQ[0].buff[TxQ[0].rptr++];
        }
   }

   UARTTxIntOff(0);
}


/* Rcv, Error Interrupt Service Routine */
void Uart0RxErrLisr(void)
{
          if(!(UARTSTAT0 & USTAT_ERROR))
          {
          if(RxQ[0].wptr+1 != RxQ[0].rptr)
          {
               RxQ[0].buff[RxQ[0].wptr++] = UARTRXB0;
               if(RxQ[0].wptr == MAXEVENT)
                    RxQ[0].wptr = 0; /*loop back*/
          }
          }

      UARTRxIntOff(0);
}
```

The UART Transmit interrupt is transmit done interrupt. So, transmitting one byte data to UART channel by interrupt mode, UART Tx interrupt test pending bit have to be set forcing to generate Tx interrupt. (see UARTTxIntOn() ,Listing 3-23). The transmit Queue data will be written to UART holding register[UARTTXHn] whenever the Tx interrupt occurred. After the written data all shifted out to UART channel, UART transmit done interrupt will be occurred.

If the UART Tx interrupt switched to On/Off repeatedly for data transfer in poll or interrupt mode, that means Tx interrupt occurred two times per each byte data. It' s caused to increase the interrupt response times. As a results, the CPU performance also can be degraded.

To avoid this situation, you have to clear the pending register at the end of the transmit interrupt service routine. As clear the pending register which is set by transmit done interrupt, the transmit done interrupt can not served. Actually, this transmit done is useless and dummy interrupt. (Listing 3-23).

Where, the SetPendingBit() is a macro function defined at interrupt handler C-code header file. (**isr.h**). Using this macro function, the UART transmit interrupt pending bit of interrupt pending test register[INTPNDTST] according to the argument value of SetPendingBit() can be set.

Listing 3-23. (uart.c)

```
void UARTTxIntOn(uint32 channel)
{
   if(channel) {
          /* Enable Interrupt */
          Enable_Int(nUART1_TX_INT);
          SetPendingBit(nUART1_TX_INT);
   }
   else {
          /* Enable Interrupt */
          Enable_Int(nUART0_TX_INT);
          SetPendingBit(nUART0_TX_INT);
   }
}

void UARTRxIntOn(uint32 channel)
{
    if(channel) {
          /* Enable Interrupt */
#ifdef KS32C50100
          /* KS32C50100 */
          Enable_Int(nUART1_RX_ERR_INT);
#else
          /* KS32C5000 */
          Enable_Int(nUART1_RX_INT);
          Enable_Int(nUART1_ERROR_INT);
#endif
    }
    else {
          /* Enable Interrupt */
#ifdef KS32C50100
          /* KS32C50100 */
          Enable_Int(nUART0_RX_ERR_INT);
#else
          /* KS32C5000 */
          Enable_Int(nUART0_RX_INT);
          Enable_Int(nUART0_ERROR_INT);
#endif
    }
}

void UARTTxIntOff(uint32 channel)
{
    if(channel) {
          /* Enable Interrupt */
          Disable_Int(nUART1_TX_INT);
      Clear_PendingBit(nUART1_TX_INT) ;
```

```
    }
    else {
            /* Disable Interrupt */
            Disable_Int(nUART0_TX_INT);
       Clear_PendingBit(nUART0_TX_INT) ;
    }
}

void UARTRxIntOff(uint32 channel)
{
    if(channel) {
            /* Disable Interrupt */
#ifdef KS32C50100
            /* KS32C50100 */
            Disable_Int(nUART1_RX_ERR_INT);
#else
            /* KS32C5000 */
            Disable_Int(nUART1_RX_INT);
            Disable_Int(nUART1_ERROR_INT);
#endif
    }
    else {
       /* Disable Interrupt */
#ifdef KS32C50100
            /* KS32C50100 */
            Disable_Int(nUART0_RX_ERR_INT);
#else
            /* KS32C5000 */
            Disable_Int(nUART0_RX_INT);
            Disable_Int(nUART0_ERROR_INT);
#endif
    }
}
```

UART Interrupted I/O functions

You can get an input character from UART channel using by get character function,i_getc(channel). (Listing 3-24). Whenever the i_getc() is called, the receive interrupt enabled by UARTRxIntOn() and the receive data queue,RxQRd(), is accessed. The Rx queue will be updated to the received new data in the Rx interrupt service routine(ISR). Receive interrupt will be disabled at the end of the Rx ISR.

Listing 3-24. (uart.c)

```
uint8 i_getc(uint32 channel)
{
   UARTRxIntOn(channel);   /* Receive interrupt ON */
   return(RxQRd(channel));   /* Read receive QUE, if there is no data in RxQ, Null will be returned */
}

/* Interrupt get string */
uint32 i_gets(uint32 channel, uint8 *s)
{
    uint32   count;
    uint8    c;

    count = 0;

    while((c = (uint8)i_getc(channel)) != CR)
    {
          count++;
          *s++ = c;
    }

    *s = (uint8)NULL;

    return(count);
}
```

For the one byte data transfer to an channel by interrupt method, I_putc() is implemented as follows: (Listing 3-25) The first, transmit complete(TC) bit of UART status register will be checked to confirm the former data all shifted out to channel. The second, transmit interrupt will be enabled when the byte data is written to transmit Queue successfully. The byte data in transmit Queue will be sent to be given channel in UART transmit interrupt service routine.

Listing 3-25. (uart.c)

```
uint32  i_putc(uint32 channel, uint8 ch)
{
   if(U_TX_COMPLETE(channel))                /* check transmit complet */
   {
        if(TxQWr(channel, ch))               /* write data to tx que */
        {
           UARTTxIntOn(channel);             /* Transmit interrupt on */
           while(!U_BUFF_EMPTY(channel));    /* wait for tx buffer empty */

           return(SUCCESS);
        }
   }

   return(ERROR);
}

/* UART Interrupt put string */
uint32  i_puts(uint32 channel, uint8 *str)
{
      uint32    i;     /* Working variable        */
      uint32    sz;

      sz = strlen((const char *)str);

       /* Loop to print out all characters.  */
       for(i=0; i < sz ; i++)
       {
            /* Call i_putc to print each character.  */
            while(!i_putc(channel, *(str + i)));
       }

       return(SUCCESS);
}

/* formatted output string */
void i_printf(char *fmt, ...)
{
   va_list argptr;
   char temp_buf[256];

   va_start(argptr, fmt);
   vsprintf(temp_buf, fmt, argptr);
   sputs((uint8 *)temp_buf);
   va_end(argptr);
}
```

StringEcho() demonstrate the usage of Interrupted UART I/O functions. (Listing 3-26).

Listing 3-26. (uart_test.c)

```
void StringEcho(uint32 channel)
{
   uint8    *bp;    /* UART test memory pointer */
   uint32   sz,i;     /* Input string size */
   uint8    ch;

   i = 0;

   do {
        i_printf("\r\r\rString echo for interrupt test.");
        i_printf("\rInput characters will be echoed on Console");
        i_printf("\r\rInput %d : ",i);

        bp  = (uint8 *)UartTxbuff;

        do {
            while((ch = i_getc(channel)) == NULL);
            *bp++ = ch;
            sz++;
        }while(ch != CR);

        bp  = (uint8 *)UartTxbuff;
        i_printf("\rEcho %d: ",i);

        while((ch = *bp++) != CR)
        {
             while(!i_putc(channel,ch));
        }

        i_printf("\rTo escape, enter ESC key : ");
        while((ch = i_getc(channel)) == NULL);
        i++;

   }while(ch != ESC);
}
```

GDMA (GENERAL DIRECT MEMORY ACCESS)

GDMA Set-Up

Figure 3-15 shows the GDMA set-up for data transfer in interrupt mode.

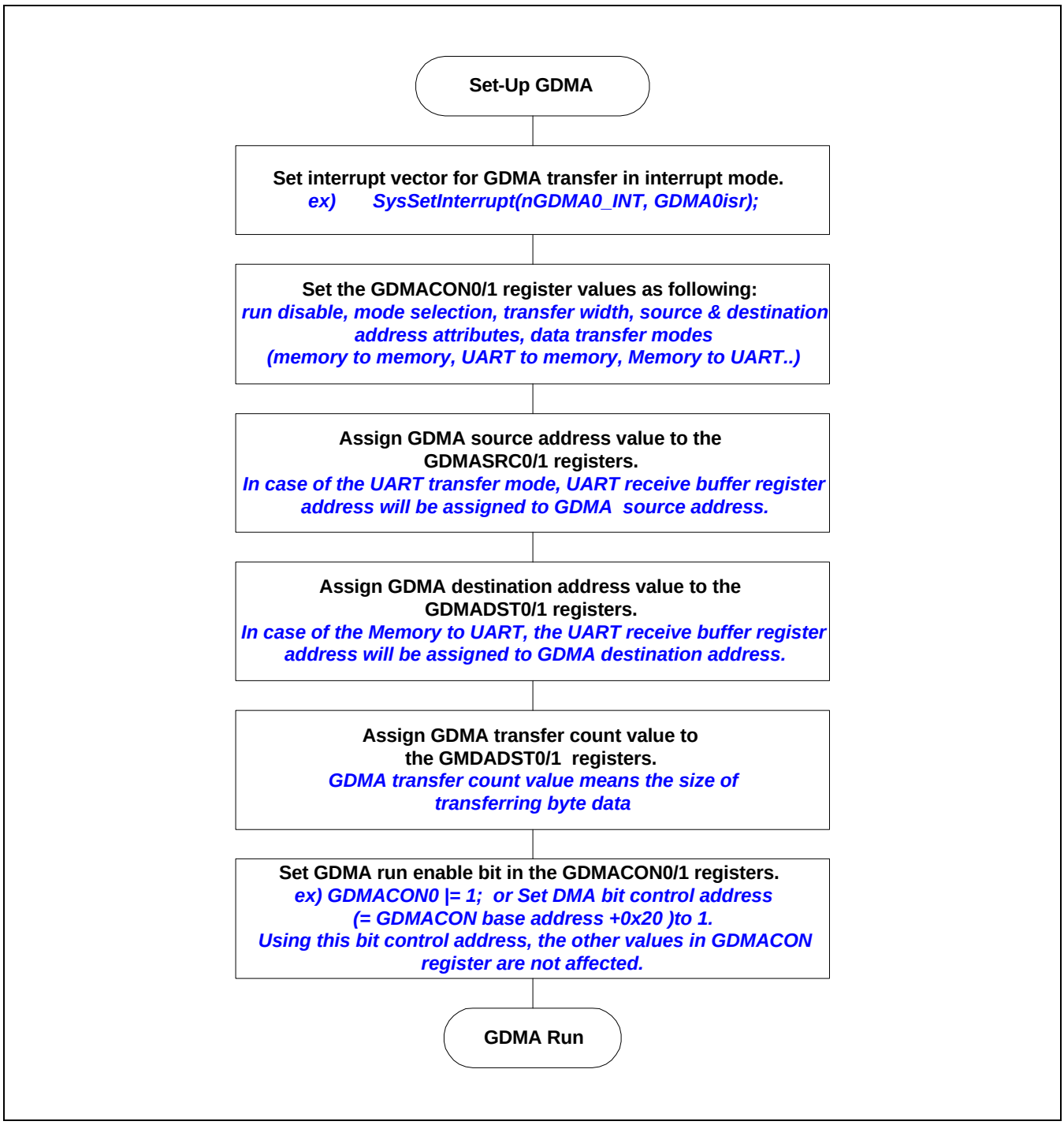


Figure 3-15. Concept Diagram for Setting Up GDMA

Listing 3-27. (dma.c)

```
/* DMA memory to memory data transfer  using Interrupt */
void dcopy(uint32 dmadst, uint32 dmasrc,int Size, int Width)
{
        uint32 DMA_CON_SET ;

        Disable_Int(nGDMA0_INT); /* Disable GDMA 0 interupt */
    SysSetInterrupt(nGDMA0_INT, GDMA0isr); /* Vector setup */
    Gdma0DoneFlag = 0;      /* GDMA transmit done flag */

        GDMASRC0    = dmasrc;
        GDMADST0    = dmadst;
        GDMACNT0    = Size;

        DMA_CON_SET = GDMA_MEM2MEM ;

        switch(Width) {
        case 0 : DMA_CON_SET |= GDMA_WIDTH_BTYE; break;
        case 1 : DMA_CON_SET |= GDMA_WIDTH_HWORD; break;
        case 2 :
        default: DMA_CON_SET |= GDMA_WIDTH_WORD; break;
        }

        Enable_Int(nGDMA0_INT);    /* Enable GDMA 0 interrupt */
        GDMACON0 = DMA_CON_SET | GDMA_RUN;  /* Start run GDMA0 */

        // By Interrupt Mode
        while(!Gdma0DoneFlag) ;
}

/* GDMA0 Transfer done interrput service routine */
void  GDMA0isr(void)
{
   Gdma0DoneFlag = 1;
}
```

32BIT TIMERS (TIMER0/1)

The concept diagram for timer set-up (Figure 3-16) shows how to configure the timer functions. These timers can be operated in interval or toggle mode. The timer output signals, TOUT0/TOUT1 can be monitored at P16[196] /P17[199]. The 32bit timer data register[TDATAn] values are automatically reloaded to timer counter register [TCNTn] whenever the timer interval expires.

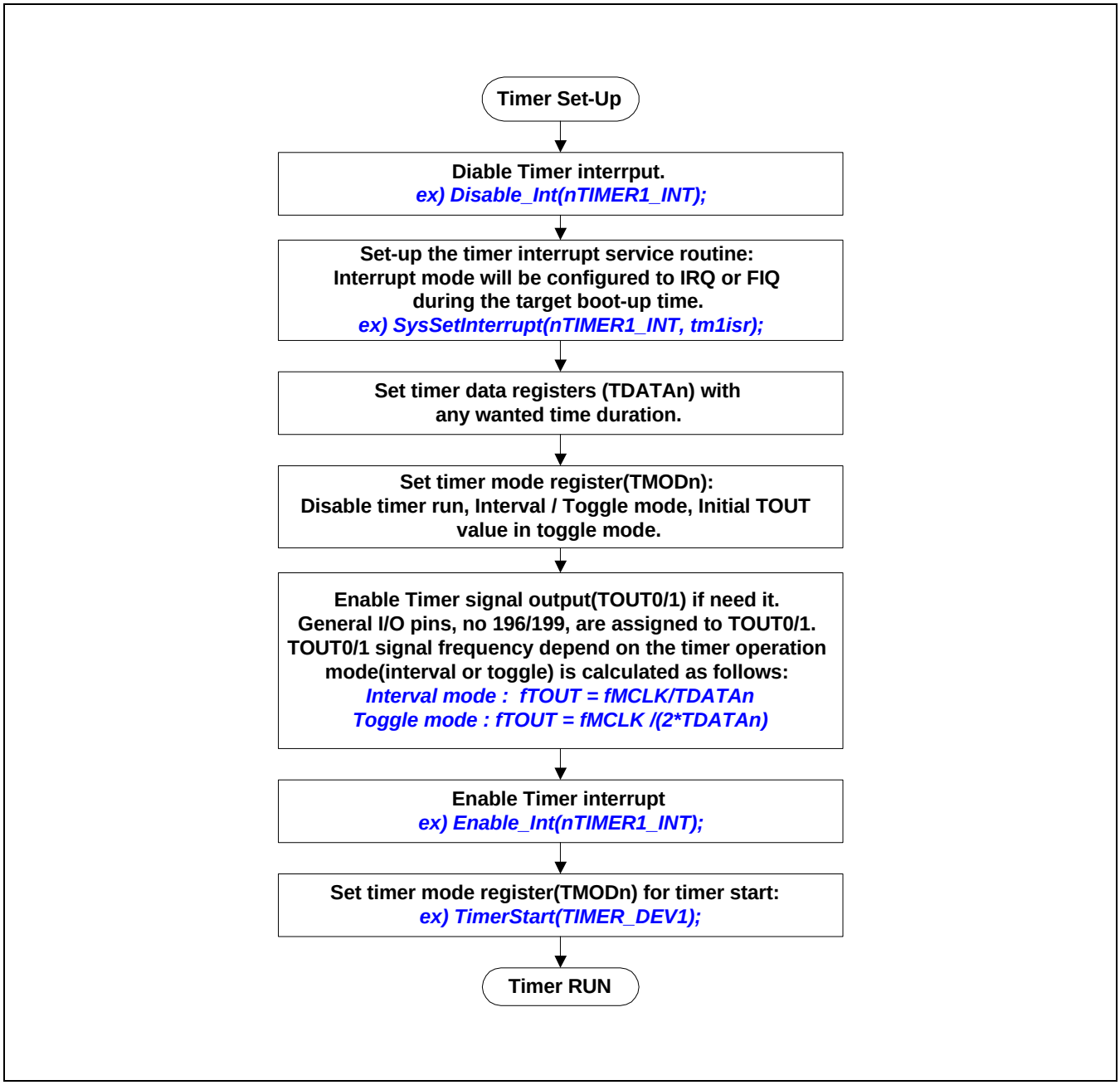


Figure 3-16. Concept Diagram for TIMER Set-Up.

Listing 3-28. (timer.h)

```
/*********************************************************/
/*          USABLE MACROS FUNCTIONS  & DATA STRUCTURES       */
/*********************************************************/

#define Timer0Stop()        (TMOD &= ~TM0_RUN)
#define Timer1Stop()        (TMOD &= ~TM1_RUN)

#define Timer0Start()       (TMOD |= TM0_RUN)
#define Timer1Start()       (TMOD |= TM1_RUN)

#define TimerStart(t)       ((t)? Timer1Start():Timer0Start())
#define TimerStop(t)        ((t)? Timer1Stop(): Timer0Stop())

#define tmDATA(t)           (t*0.001*fMCLK-1)  // t is time tick,unit[ms]
#define t_data_ms(t)        (t*0.001*fMCLK-1)  // t is time tick,unit[ms]
#define t_data_us(t)        (t*0.000001*fMCLK-1)  // t is time tick,unit[us]

typedef struct {
    void (*TIMER_Lisr)();   /* TIMER Interrupt Function Pointer */
    uint32 TM_CHANNEL;      /* TIMER DEVICE */
    uint32 TM_MODE;         /* Timer mode register */
    uint32 TM_DATA;         /* Timer data,timer range is 1~0xffffffff */
    uint32 TM_OUT_PORT;     /* Enable timer output port */
}TM_PARAM;


typedef struct {
    volatile unsigned int tm_sec;
    volatile unsigned int tm_min;
    volatile unsigned int tm_hour;
    volatile unsigned int tm_mday;
    volatile unsigned int tm_mon;
    volatile unsigned int tm_year;
}TIME;
```

Listing 3-29. (timer.c)

```
/*************************************************************************/
/*                                                                       */
/* NAME : tm_init(timer device, time periode)                            */
/*                                                                       */
/* FUNCTIONS : Initialize the TIMER0,1.                                  */
/*                                                                       */
/* EXAMPLES :                                                            */
/*           tm_init(TIMER_DEV0,ONE_SECOND/TICKS_PER_SECOND);            */
/*                                                                       */
/*             Where, the TIMER_DEV0 means timer0.                       */
/*                    ONE_SECOND = 1000,                                 */
/*                    TICKS_PER_SECOND = 100,                            */
/*              then timer0 timer periode become to 10ms.                */
/*                                                                       */
/* VARIABLES USED                                                        */
/*                                                                       */
/*                                                                       */
/* HISTORY                                                               */
/*                                                                       */
/*       NAME          DATE                REMARKS                     */
/*                                                                       */
/*      in4maker     06-07-1999     Created initial version 1.0         */
/*                                                                       */
/*************************************************************************/
void tm_init(int TIMER_DEV, int t)
{

  if(TIMER_DEV)  /* for TIMER 1 */
  {
      Disable_Int(nTIMER1_INT);
          SysSetInterrupt(nTIMER1_INT, tm1isr);

      //IOPDATA &= ~(1<<17);    /* Output 0 to P17(TOUT1) */
      //IOPMOD  |= (1<<17);     /* Enable P17 to output port */
                        /* TOUT1 will be cleared */
      TDATA1 = t_data_ms(t);    /* unit is [ms] */
      TCNT1 = 0x0;
      TMOD  = TM1_TOGGLE;       /* Toggle pulse will be out to port */
      Enable_Int(nTIMER1_INT);  /* Timer interrupt enable */
      //IOPCON = (1<<30);       /* Timer1 output(TOUT1)enable */
  }
  else  /* for TIMER0 */
  {
      Disable_Int(nTIMER0_INT);
          SysSetInterrupt(nTIMER0_INT, tm0isr);

      //IOPDATA &= ~(1<<16);    /* Output 0 to P16(TOUT0) */
      //IOPMOD  |= (1<<16);     /* Enable P16 to output port */
                        /* TOUT0 will be cleared */
      TDATA0 = t_data_ms(t);
```

```
        TCNT0 = 0x0;
        TMOD  = TM0_TOGGLE;
        Enable_Int(nTIMER0_INT);
        //IOPCON = (1<<29);       /* Timer0 output(TOUT0)enable */
    }
}

/*************************************************************************/
/*                                                      */
/* NAME : tm0isr()                                              */
/*                                                      */
/* FUNCTIONS : Timer0 interrupt service routine.                        */
/*                                                      */
/*************************************************************************/
void tm0isr(void)
{
    clk_tick0++;

    if(clk_tick0 == TICKS_PER_SECOND)
    {
        clk_tick0 = 0;
        if(tm0.tm_sec++ > 59)
        {
            tm0.tm_sec = 0;
            if(tm0.tm_min++ > 59)
            {
                tm0.tm_min = 0;
                if(tm0.tm_hour++ > 23)
                {
                    tm0.tm_hour = 0;
                    if(tm0.tm_mday++ > 30)
                    {
                        tm0.tm_mday = 0;
                        if(tm0.tm_mon++ > 11)
                        {
                            tm0.tm_mon = 0;
                            tm0.tm_year++;
                        }
                    }
                }
            }
        }
        /* 4 means digit number for LED display */
        IOPDATA = ~(1<<tm0.tm_sec%4);
    }
}

            - - - - - - - - - - - -
```

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.

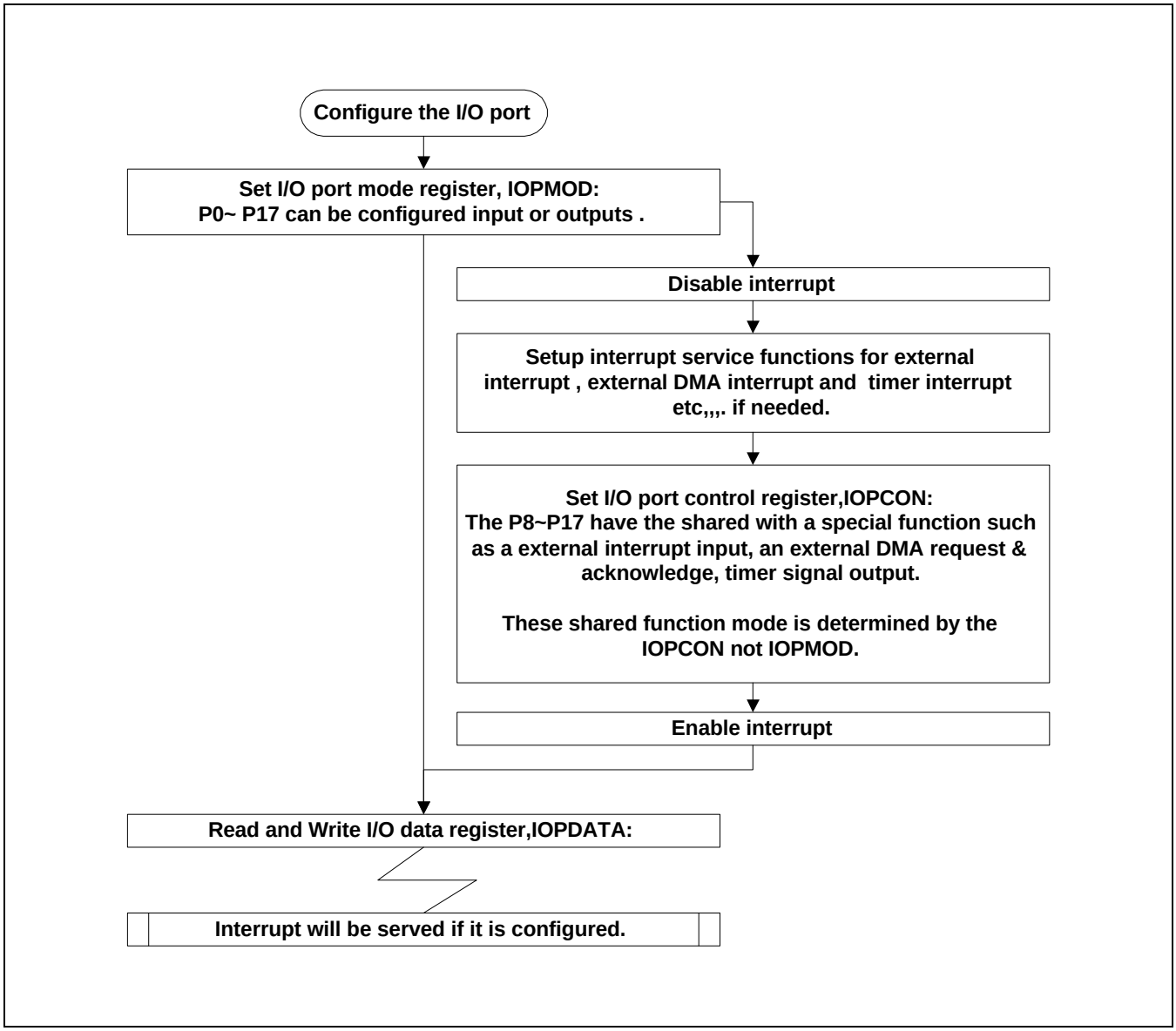


Figure 3-17. Concept Diagram for the Configuring of I/O Port

HIGH-LEVEL DATA LINK CONTROLLER FOR KS32C5000(A)

HDLC Diagnostic Code Function for KS32C5000(A)

The diagnostic source code for the HDLC (High-Level Data Link Controller) is composed of four files, HDLC.H, HDLC.C, HDLCINIT.C, and HDLCLIB.C.

HDLC.H: Definition file for HDLC diagnostic code.

HDLC.C: The main functions for HDLC diagnostic test.

HDLCINIT.C: Initialize HDLC and HDMA controller for normal operating environment, each interrupt service routine.

HDLCLIB.C: The library functions for HDLC diagnostic code.

Definitions of HDLC for KS32C5000(A)

Define macro function for each HDLC controller control and status register address, and some simple function for control HDLC controller. The definition source code is described in **Listing 3-30**.

Listing 3-30. Definition for HDLC for KS32C5000(A) (HDLC.H)

```
// Macro function for define HDLC Registers
#define HCON0(channel)                  (VPint(Base_Addr+0x7000 + channel*0x1000))
#define HCON1(channel)                  (VPint(Base_Addr+0x7004 + channel*0x1000))
#define HSTAT(channel)                 (VPint(Base_Addr+0x7008 + channel*0x1000))
#define HINTEN(channel)                (VPint(Base_Addr+0x700c + channel*0x1000))
#define HTXFIFOC(channel)              (VPint(Base_Addr+0x7010 + channel*0x1000))
#define HTXFIFOT(channel)              (VPint(Base_Addr+0x7014 + channel*0x1000))
#define HRXFIFO(channel)               (VPint(Base_Addr+0x7018 + channel*0x1000))
#define HSADR(channel)                  (VPint(Base_Addr+0x701c + channel*0x1000))
#define HBRGTC(channel)                (VPint(Base_Addr+0x7020 + channel*0x1000))
#define HPRMB(channel)                  (VPint(Base_Addr+0x7024 + channel*0x1000))
#define HDMATXMA(channel)              (VPint(Base_Addr+0x7028 + channel*0x1000))
#define HDMARXMA(channel)              (VPint(Base_Addr+0x702c + channel*0x1000))
#define HDMATXCNT(channel)             (VPint(Base_Addr+0x7030 + channel*0x1000))
#define HDMARXCNT(channel) (VPint(Base_Addr+0x7034 + channel*0x1000))
#define HDMARXBCNT(channel)         (VPint(Base_Addr+0x7038 + channel*0x1000))

// Macro function for HDLC controller control
#define HDLC_Tx_Enable(channel)       HCON0(channel) |= TxEN
#define HDLC_Rx_Enable(channel)       HCON0(channel) |= RxEN
#define HDMA_Tx_Enable(channel)       HCON0(channel) |= DTxEN
#define HDMA_Rx_Enable(channel)       HCON0(channel) |= DrxEN
#define HDLC_Loopback_enable(channel)        HCON1(channel) |= TxLOOP
#define HDLC_Loopback_disable(channel)       HCON1(channel) &= ~TxLOOP
```

HDLC Initialize for KS32C5000(A)

The HDLC and HDMA should be initialized before getting into operation. HdlcInitialize(), and HdlcPortInit() function is used for this initialization. In the following, describe the contents of this function.

In the high-speed operation of HDLC, HDMA can be used for transferring and receiving data to memory and transfer the transmit data to HDLC. The HDLC can support Full duplex operation, and speed up to 4Mbps using an external/internal clock. When you use HDLC controller to your system, You need to set the HDLC, and HDMA controller to work properly.

When we setup HDLC controller, we use Device_Entry structure, The Device_Entry structure has all parameter that should be setup for normal operation. The Device_Entry structure is described in **Listing 3-31**.

Listing 3-31. Device_Entry structure for KS32C5000(A) (HDLC.H)

```
typedef struct HDLC_Device {
        U32 HDLC_Port ;                 // HDLC port number
        U32 HDLC_Baud ;                 // HDLC baudrate
        U32 HDLC_Data_format ;    // HDLC Data format, NRZ,NRZI,FM0,FM1,Menchester
        U32 HDLC_Tx_Mode ;  // HDLC Tx DMA or Interrupt mode
        U32 HDLC_Rx_Mode ;  // HDLC Rx DMA or Interrupt mode
        U32 HDLC_Tx_Output_Clk ;   // Tx clock PIN output source, Rx_Clock,BRGOUT1,2,3
                                   // DPLLOUTT, DPLLOUTR
        U32 HDLC_Tx_Clk ;             // Tx clock source,TxC pin, RxC pin, DPLLOUTT, BRGOUT1,2,3
        U32 HDLC_Rx_Clk ;     // Rx clock source,TxC pin, RxC pin, DPLLOUTT, BRGOUT1,2,3
        U32 HDLC_Tx_Clk_Pos ;         // Tx clock polarity, only for KS32C5000A
        U32 HDLC_Rx_Clk_Pos ;         // Rx clock polarity, only for KS32C5000A
        U32 HDLC_Station_Addr ;
} HDLC_Device_Entry ;
```

The initialize parameter for each HDLC controller is described in source code **Listing 3-32**. After initialize the HDLC parameter, the HdlcPortInit() function is used for initialize each port with initialization parameter value. And, We also need to setup the interrupt vector table for each HDLC controller Interrupt source.

Listing 3-32. HdlcInitialize( ) function (HDLCINIT.C)

```
/*
 *  Function : HdlcInitialize
 *  Description : HDLC controller initialize
 */
void HdlcInitialize(void)
{
        HDLC_Device_Entry  *Dev_entry[HDLCPORTNUM] ;
        int        channel;

        Disable_Int(nHDLCA_INT);
        Disable_Int(nHDLCB_INT);

        // Step 1. Set HDLC channel A initialize parameter
        Dev_entry[HDLCA] = (HDLC_Device_Entry *)&HDLC_Dev[HDLCA] ;
        Dev_entry[HDLCA]->HDLC_Port                = HDLCA ;
        Dev_entry[HDLCA]->HDLC_Baud                = 64000 ;
        Dev_entry[HDLCA]->HDLC_Data_format         = DF_NRZ ;
        Dev_entry[HDLCA]->HDLC_Tx_Mode             = MODE_DMA ;
        Dev_entry[HDLCA]->HDLC_Rx_Mode             = MODE_DMA ;
        Dev_entry[HDLCA]->HDLC_Tx_Output_Clk       = TxOCLK_BRG1 ;
        Dev_entry[HDLCA]->HDLC_Tx_Clk              = TxCLK_BRG1 ;
        Dev_entry[HDLCA]->HDLC_Rx_Clk              = RxCLK_RXC ;
        Dev_entry[HDLCA]->HDLC_Station_Addr        = HDLC_STATION_ADDR ;

        // Tx clock polarity, only for KS32C5000A,
        // when set 0 the polarity is same as KS32C5000
        Dev_entry[HDLCA]->HDLC_Tx_Clk_Pos          = TxCNEG ;

        // Rx clock polarity, only for KS32C5000A,
        // when set 1 the polarity is same as KS32C5000
        Dev_entry[HDLCA]->HDLC_Rx_Clk_Pos          = RxCPOS ;


        // Step 2. Set HDLC channel B initialize parameter
        Dev_entry[HDLCB] = (HDLC_Device_Entry *)&HDLC_Dev[HDLCB] ;
        Dev_entry[HDLCB]->HDLC_Port                = HDLCB ;
        Dev_entry[HDLCB]->HDLC_Baud                = 64000 ;
        Dev_entry[HDLCB]->HDLC_Data_format         = DF_NRZ ;
        Dev_entry[HDLCB]->HDLC_Tx_Mode             = MODE_DMA ;
        Dev_entry[HDLCB]->HDLC_Rx_Mode             = MODE_DMA ;
        Dev_entry[HDLCB]->HDLC_Tx_Output_Clk       = TxOCLK_BRG1 ;
        Dev_entry[HDLCB]->HDLC_Tx_Clk              = TxCLK_BRG1 ;
        Dev_entry[HDLCB]->HDLC_Rx_Clk              = RxCLK_RXC ;
        Dev_entry[HDLCB]->HDLC_Station_Addr        = HDLC_STATION_ADDR ;

        // Tx clock polarity, only for KS32C5000A,
        // when set 0 the polarity is same as KS32C5000
        Dev_entry[HDLCB]->HDLC_Tx_Clk_Pos          = TxCNEG ;

        // Rx clock polarity, only for KS32C5000A,
```

```
        // when set 1 the polarity is same as KS32C5000
        Dev_entry[HDLCB]->HDLC_Rx_Clk_Pos       = RxCPOS ;


        // Step 3. Initialize each HDLC port
        for(channel=HDLCA ; channel<=HDLCB ; channel++)
        {
            if ( !HdlcPortInit(Dev_entry[channel]) )
                Print("\n HDLC channel [%d] initialize error",channel) ;
        }

        // Step 4. Setup Interrupt for HDLC
        SysSetInterrupt(nHDLCA_INT, HDLCA_isr );
        SysSetInterrupt(nHDLCB_INT, HDLCB_isr );

        // Step 5. Enable HDLC interrupt
        Enable_Int(nHDLCA_INT);
        Enable_Int(nHDLCB_INT);
}
```

After setup HDLC parameter, the main HDLC initialization is performed by HdlcPortInit() function. The detail of HDLC initialization function is described in source code **Listing 3-33**, and **Figure 3-19**.

Listing 3-33. HdlcPortInit() Function for KS32C5000(A) (HDLCINIT.C)

```
/*
 *  Function : HdlcPortInit
 *  Description : HDLC Port initialize
 */
int HdlcPortInit(HDLC_Device_Entry *Device)
{
        U32     TxInterruptEnableFlag ;
        U32     RxInterruptEnableFlag ;
        U32     channel ;
        channel = Device->HDLC_Port ;

        // Step 1. Reset HDLC Controller and HDLC DMA
        HCON0(channel) = TxRS | RxRS | DTxRS | DRxRS ;

        // Step 2. Set Station address
        // This register only used when enable RxASEN. Now, we didn't use
        HSADR(channel) = Device->HDLC_Station_Addr ;

        // Step 3. Set Baud Rate Generater constant value
        HBRGTC(channel)= fMCLK/(2*Device->HDLC_Baud)-1;

        // Step 4. Set Preamble value for NRZ code
        HPRMB(channel) = HDLC_PREAMBLE ;

        // Step 5. Set HDLC Control Register 1
        HCON1(channel) = gHCon1 | \
```

```
                        Device->HDLC_Data_format |\
                        Device->HDLC_Tx_Output_Clk |\
                        Device->HDLC_Tx_Clk |\
                        Device->HDLC_Rx_Clk ;

        // Step 6. Set HDLC Control Register 0
        // Baud-rate Generator Enable, DPLL Enable, Tx/Rx Enable
        HCON0(channel) = gHCon0 | \
                        Device->HDLC_Tx_Clk_Pos | \
                        Device->HDLC_Rx_Clk_Pos ;

        // Step 7. Tx, and Rx Buffer descriptor initialize
        TxBDInitialize(channel) ;
        RxBDInitialize(channel) ;

        // Step 8. Enable HDLC Transmit controller
        HDLC_Tx_Enable(channel);

        // Step 9. Setup HDLC and HDMA for receive operation
        HDLC_Rx_init(channel) ;

        // Step 10. Set HDLC Interrupt enable register
        if ( Device->HDLC_Tx_Mode == MODE_DMA )
                TxInterruptEnableFlag = DTxIE | TxUIE | DTxABIE | \
                                        DTxSTOPIE | TxUIE ;
        else
                TxInterruptEnableFlag = TxUIE ;

        if ( Device->HDLC_Rx_Mode == MODE_DMA )
                RxInterruptEnableFlag = DRxIE | RxABIE | RxFERRIE | RxOVIE | \
                                        DRxSTOPIE | DRxABIE ;
        else
                RxInterruptEnableFlag = RxABIE | RxFERRIE | RxOVIE | RxFAIE | RxFVIE ;

        gHintEn |= TxInterruptEnableFlag | RxInterruptEnableFlag ;
        HINTEN(channel) = gHintEn ;

        // Step 11. Clear HDLC Status
        HSTAT(channel) |= HSTAT(channel) ;
    return 1 ;
}
```

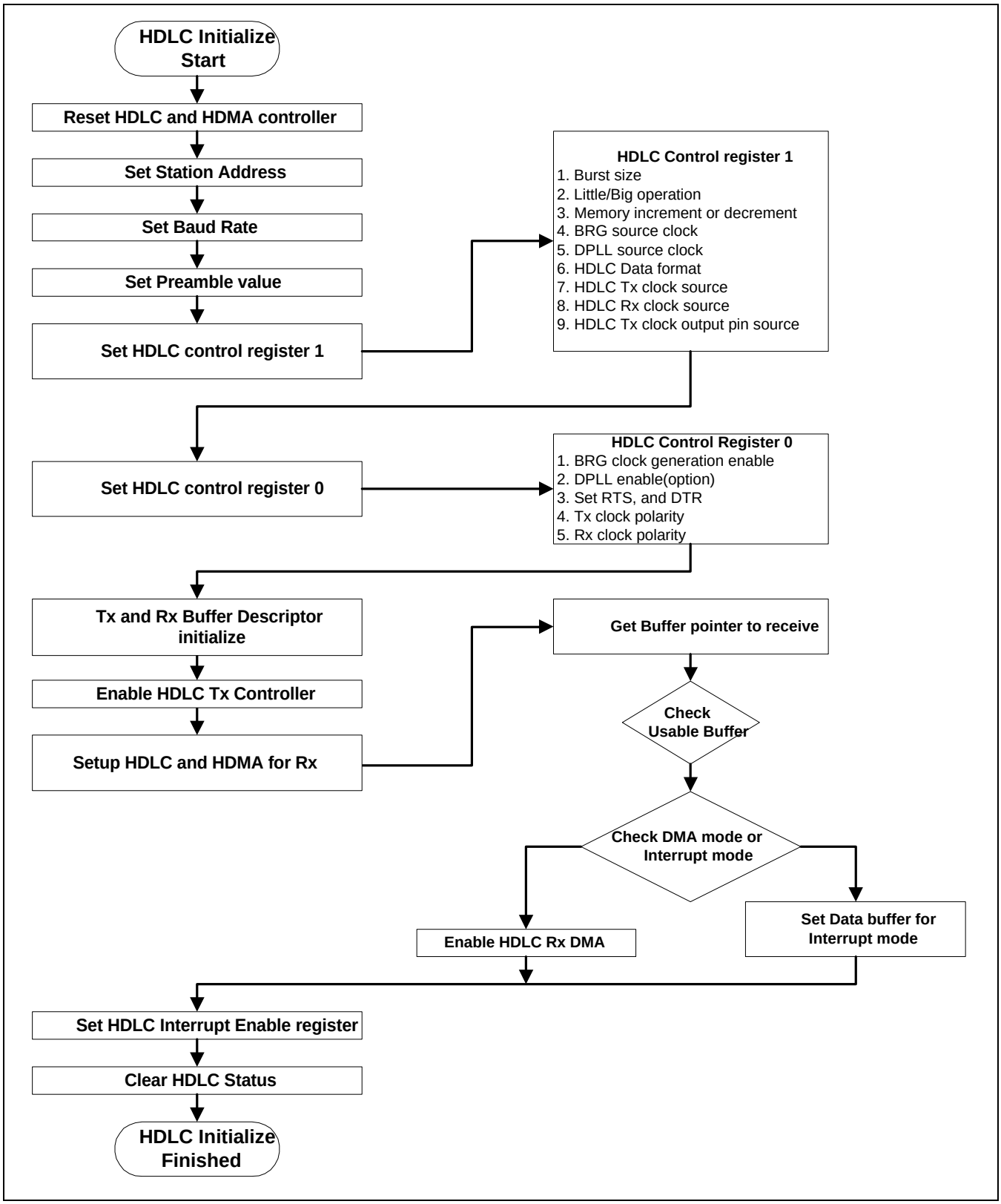


Figure 3-19. HDLC Initialization flow

The each step of HDLC initializing function is described as followings.

STEP 1. Reset HDLC Controller and HDLC DMA

First of all, it is necessary to reset HDLC and DMA controller, which initialize registers as default values.

STEP 2. Set Station Address

The Station address can be used when RxASEN bit is setting to address search.

STEP 3. Set Baud Rate Generater constant value to BRGTC

STEP 4. Set Preamble value

STEP 5. Set HDLC Control Register 1 (**Table 3-5)**

Table 3-5. HDLC Control Register 1 Initialize

Control Bit Name	Description	In Sample Code	
		Control	Behavior
DF	Selecting Data Format NRZ,NRZI,FM0,FM1, and Manchester	NRZ	NRZ data format will be used for Data Encoding/Decoding for each channel.
TxLittle	Selecting Endian mode Select whether data bytes are swapped or not between system bus and HDLC Tx/Rx FIFO	TxBig	The transmitted data will be a Big Endian format.
RxLittle		RxBig	We assume the data which will be received as Big Endian format.
RxCLK	Selecting Rx clock Rx clock can be selected from BRG, DPLLOUTR, or external TxC/RxC pin.	RxCLK_RxC	External clock to RxC pin is selected as Rx clock.
TxCLK	Selecting Tx clock Tx clock can be selected from BRG, DPLLOUTT, or external TxC/RxC pin.	TxCLK_BRG1	BRGOUT2 clock generated by BRG block is selected as Tx clock.
TXCOPS	Selecting TxC pin as Output	TxOCLK_BRG1	BRG1 clock is output tot TXC pin.
BRGCLK	Internal BRG clock for Tx/Rx clock BRG source clock can be select between MCLK and RxC pin. When use BRG clock, you should enable BRGEN bit on HCON0 register	BRGCLK_MCLK	System Main clock selected as BRG source clock.

In the HDLC, BRG output, or DPLL output clock can be used (BRG or DPLL should be enabled). BRG output clock is depend on the Time Constant value which should be written to HBRGTC register. When use BRG clock, BRGOUT1 is faster than 32 times than BRGOUT3, and 16 times than BRGOUT2 in same Time Constant value.

When set the HDLC clock, another consideration is data type. If incoming data type is NRZ or NRZI, the source clock of the DPLL must be 32 times faster than RxD. In FM or Menchester, 16 times faster clock is needed. Using the DPLL output clock, DPLLOUTT for Tx clock and DPLLOUTR fro Rx clock can be selected respectively.
The HDLC clock diagram is depicted in **Figure 3-20**.

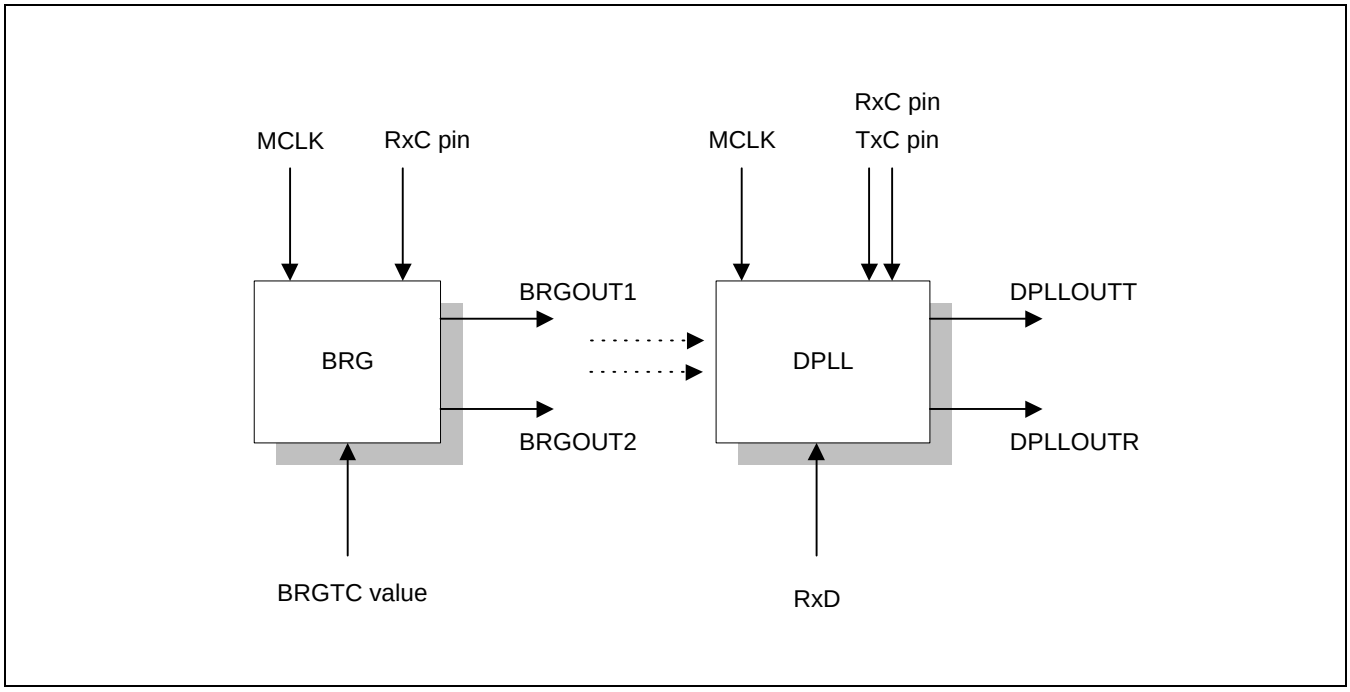


Figure 3-20. HDLC Internal clock Block Diagram

STEP 6. Set HDLC Control Register 0

In the HDLC Control Register 0, BRG(Baud Rate Generator) counter should be enabled when use BRG clock. And, In the KS32C5000A has some different TxC, and RxC pin input polarity from KS32C5000. So, when you use KS32C5000A, you should set the clock polarity value.

[Note] Hardware Flow Control

HDLC control register 0(HCON0) register has two bits to manage Hardware Flow Control. TxDTR bit directly affects the nDTR pin output state if TxEN bit is enabled. When TxDTR bit is cleared, nDTR goes HIGH. And AutoEN bit controls the function of nDCD and nCTS.

STEP 7. Tx, and Rx Buffer Descriptor Initialize.

This buffer descriptor is used for receive and transmit data buffer, all receive and transmit operation use this buffer descriptor. The buffer descriptor structure is described in **Listing 3-34**, and **Figure 3-21**.

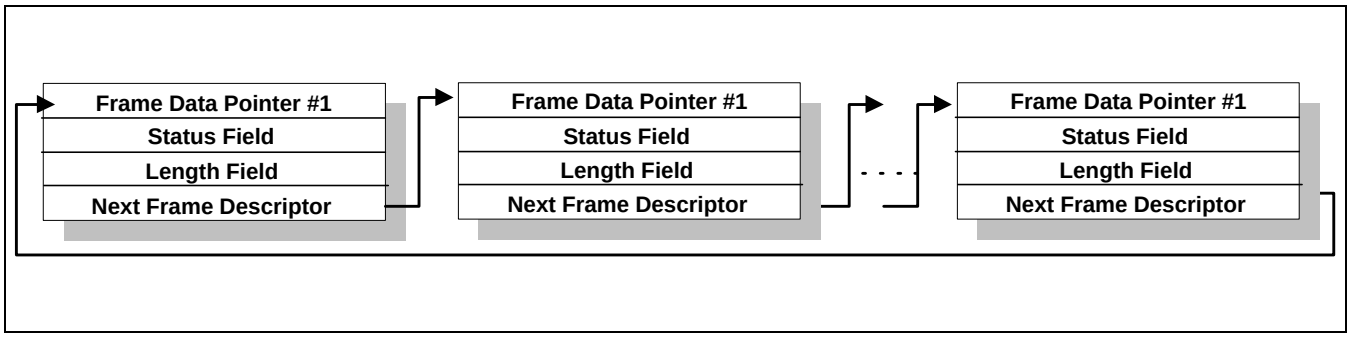


Figure 3-21. HDLC Buffer Descriptor Structure for KS32C5000(A)

Listing 3-34. HDLC Buffer descriptor structure for KS32C5000(A) (HDLCINIT.C)

```
// Tx/Rx Buffer descriptor structure
typedef struct BD {
        U32 BufferDataPtr;
        U32 StatusField ;
        U32 LengthField ;
        U32 NextBufferDescriptor ;
} sBufferDescriptor;
```

The HDLC Tx/Rx buffer descriptor, and data buffer area should be non-cacheable, because HDLC DMA can update the value, so when we initialize buffer descriptor, use NonCache(= 0x4000000) for non-cacheable access. The **Listing 3-35**, and **Listing 3-36** show the detail operation of setup HDLC Tx/Rx buffer descriptor.
The default owner of transmit HDLC buffer descriptor is CPU, and the default owner of receive HDLC buffer descriptor owner is DMA,
After receive frame, in the interrupt service routine for receive operation, buffer descriptor owner is changed to CPU owner, then it can be processed by CPU(user receive operation), after process this received frame, CPU change buffer owner to DMA again to receive next frame.
When transmit a frame, CPU change the owner to DMA after initialize frame data to buffer, after transmit a frame, in the interrupt service routine for transmit operation, the owner is changed to CPU again to use next frame transmit buffer.

Listing 3-35. TxBDInitialize() function for KS32C5000(A) (HDLCINIT.C)

```
/*
 * Function : void TxBDInitialize(U32 channel) ;
 * Description : Initialize Tx buffer descriptor area-buffers.
 */
void TxBDInitialize(U32 channel)
{
        sBufferDescriptor *pBufferDescriptor;
        sBufferDescriptor *pStartBufferDescriptor;
        sBufferDescriptor *pLastBufferDescriptor = NULL;
        U32 BufferDataAddr;
        U32 i;

        // Get Buffer descriptor's base address.
        // +0x4000000 is for setting this area to non-cacheable area.
        gCTxBDPtr[channel] = (U32)TxBDABase[channel] + NonCache ;
        pCTxBDPtr[channel] = (U32)TxBDABase[channel] + NonCache ;

        // Get Transmit buffer base address.
        HDMATXMA(channel) = BufferDataAddr = (U32)TxBABase[channel] + NonCache ;

        // Generate linked list.
        pBufferDescriptor = (sBufferDescriptor *) gCTxBDPtr[channel];
        pStartBufferDescriptor = pBufferDescriptor;

        for(i=0; i < MaxTxBufferDescriptor; i++)  {
                if(pLastBufferDescriptor == NULL)
                        pLastBufferDescriptor = pBufferDescriptor;
                else    pLastBufferDescriptor->NextBufferDescriptor = (U32)pBufferDescriptor;

                pBufferDescriptor->BufferDataPtr =
                        (U32)(BufferDataAddr & BOwnership_CPU);
                pBufferDescriptor->StatusField = (U32)0x0;
                pBufferDescriptor->LengthField = (U32)0x0;
                pBufferDescriptor->NextBufferDescriptor = NULL;

                pLastBufferDescriptor = pBufferDescriptor;
                pBufferDescriptor++;
                BufferDataAddr += sizeof(sHDLCFrame);
        } // end for loop

        // Make Buffer descriptor to ring buffer type.
        pBufferDescriptor--;
        pBufferDescriptor->NextBufferDescriptor = (U32)pStartBufferDescriptor;
}
```

Listing 3-36. RxBDInitialize() function for KS32C5000(A) (HDLCINIT.C)

```
/*
 * Function : void RxBDInitialize(U32 channel) ;
 * Description : Initialize Rx buffer descriptor area-buffers.
 */
void RxBDInitialize(U32 channel)
{
        sBufferDescriptor *pBufferDescriptor;
        sBufferDescriptor *pStartBufferDescriptor;
        sBufferDescriptor *pLastBufferDescriptor = NULL;
        U32 BufferDataAddr;
        U32 i;

        // Get Buffer descriptor's base address.
        // +0x4000000 is for setting this area to non-cacheable area.
        gCRxBDPtr[channel] = (U32)RxBDABase[channel] + NonCache ;
        pCRxBDPtr[channel] = (U32)RxBDABase[channel] + NonCache ;

        // Get Transmit buffer base address.
        HDMARXMA(channel) = BufferDataAddr = (U32)RxBABase[channel] + NonCache ;

        // Generate linked list.
        pBufferDescriptor = (sBufferDescriptor *) gCRxBDPtr[channel];
        pStartBufferDescriptor = pBufferDescriptor;

        for(i=0; i < MaxRxBufferDescriptor; i++)  {
                if(pLastBufferDescriptor == NULL)
                        pLastBufferDescriptor = pBufferDescriptor;
                else    pLastBufferDescriptor->NextBufferDescriptor = (U32)pBufferDescriptor;

                pBufferDescriptor->BufferDataPtr =
                        (U32)(BufferDataAddr | BOwnership_DMA | NonCache );
                pBufferDescriptor->StatusField = (U32)0x0;
                pBufferDescriptor->LengthField = (U32)0x0;
                pBufferDescriptor->NextBufferDescriptor = NULL;

                pLastBufferDescriptor = pBufferDescriptor;
                pBufferDescriptor++;
                BufferDataAddr += sizeof(sHDLCFrame);
        } // end for loop

        // Make Buffer descriptor to ring buffer type.
        pBufferDescriptor--;
        pBufferDescriptor->NextBufferDescriptor = (U32)pStartBufferDescriptor;
}
```

STEP 8. Enable HDLC Transmit controller

STEP 9. Setup HDLC and HDMA for receive operation

This step setup the HDLC and HDMA can receive frame to reserved buffer. The step of setup flow of HDLC and HDMA can receive is described in below.

1. Get HDLC device entry pointer

At the first time get the HDLC device entry point to control HDLC buffer.

2. Get receive buffer pointer from global variable

Get the pointer of receive buffer descriptor, and data buffer from global variable that has current buffer descriptor pointer

3. Check Rx DMA ownership

Check ownership of buffer descriptor that from current buffer descriptor pointer, If ownership is CPU, then current buffer can used for received buffer, but if this buffer pointer is owned by DMA, then just exit this initialize routine with error information.

4. Clear Ownership bit for DMA

If current buffer can we use, then we can get data buffer pointer, but when set the owner bit, this is not a visible address area, The clear operation is only need to get data pointer address.

5. Setup Rx DMA and Data buffer

If receive mode is DMA mode, then setup receive HDMA to can receive a frame from HDLC controller and enable the HDLC receive operation, but if receive mode is interrupt method, then we only get the data pointer that can be used for temporary received data buffer.

6. Receive enable

After setup HDLC and HDMA controller, we can enable HDLC controller can receive the incomming data.

The source code for HDLC and HDMA initialize function for receive operation is described in **Listing 3-37**.

Listing 3-37. HDLC_Rx_init() function for KS32C5000(A) (HDLCINIT.C)

```
/*
 *  Function : HDLC_Rx_init
 *  Description : HDLC Rx initialize
 *  return 1 : HDLC receive initialize ok
 *  return 0 : HDLC receive initialize fail
 */
int HDLC_Rx_init(U32 channel)
{
        sBufferDescriptor       *CRxBDPtr ;
        HDLC_Device_Entry       *Dev ;
        U32                     DataBuffer;

        // Step 1. Get HDLC device entry pointer
        Dev = (HDLC_Device_Entry *)&HDLC_Dev[channel] ;

        // Step 2. Get receive buffer pointer from global variable
        CRxBDPtr = (sBufferDescriptor *)gCRxBDPtr[channel] ;
        DataBuffer = (U32)CRxBDPtr->BufferDataPtr ;

        // Step 3. Check Rx DMA ownership
        if ( ((U32)(CRxBDPtr->BufferDataPtr) & BOwnership_DMA) )  {

                // Step 4. Clear Ownership bit for DMA
                DataBuffer = (U32)(CRxBDPtr->BufferDataPtr & BOwnership_CPU ) ;

                // Step 5. Setup Rx DMA and Data buffer
                if ( Dev->HDLC_Rx_Mode == MODE_DMA ) {
                        HDMARXBCNT(channel)         = 0x0  ;
                        HDMARXMA(channel)   = DataBuffer ;
                        HDMARXCNT(channel)          = sizeof(sHDLCFrame) ;
                        HDMA_Rx_Enable(channel) ;
                }
                else {
                        ModeInt_RxDataBuffer[channel] = (U32 *)DataBuffer ;
                        ModeInt_RxDataSize[channel] = 0 ;
                }

                // Step 6. Receive enable
                HDLC_Rx_Enable(channel) ;

                return 1 ;
        }
        else return 0 ; // buffer full state, so can't use this buffer before get used
}
```

STEP 10. Set HDLC Interrupt enable register

HDLC has many interrupt source, HDLC Tx, HDLC Rx, HDMA Tx, and HDMA Rx interrupt source, this routine enable each interrupt source for each Tx/Rx method.

STEP 11. Clear HDLC Status register bit value.

NOTE: The Way to Use Status Register

There are three kinds of bits in status register. One sort of these bits are cleared automatically when this bit indicating status is cleared. These status bits are TxFA, TxCTS, RxFA, and RxDCD. The second kinds of bits are cleared by reading FIFO or this bit, the RxFAP, FxAERR,RxFV, and RxFERR bits are cleared by reading RxFIFO in interrupt mode, and cleared by read this bit in DMA mode. And, the other all status is cleared by CPU writing 1.

Transmit HDLC frame with KS32C5000(A)

After setting all control register and HDMA transmit buffer descriptor, you can transmit packet. At the first time you should prepare frame to transmit. The sample code for prepare frame is shown in **Listing 3-38**.

Listing 3-38. Transmit_Frame() function for KS32C5000(A) (HDLCLIB.C)

```
.....
        // Step 1. Set destination address fields
        FrameBuffer.Header.Address[0] = 0x12 ;
        FrameBuffer.Header.Address[1] = 0x34 ;
        FrameBuffer.Header.Address[2] = 0x45 ;
        FrameBuffer.Header.Address[3] = 0x67 ;

        // Step 2. Set Control field
        FrameBuffer.Header.Control[0] = 0xff ;

        // Step 3. Set HDLC Frame Data
        for (i=0 ; i< (Size-(sizeof(sHDLCHeader))) ; i++)
                FrameBuffer.Information[i]  = (U8)(i & 0xFF) ; // sample code, just has sequence
...
        // Step 4. Send HDLC frame
        if (!SendHdlcFrame(channel,(U8 *)&FrameBuffer,Size) )
                        Print("\n HDLC %d Send error",channel) ;
.....
```

The SendHdlcFrame() function is used for transmit HDLC frame. The source code of SendHdlcFrame() function is listed in **Listing 3-39**, and the transmit flow of HDLC frame is depicted in **Figure 3-22**.
In the DMA mode, HDLC transmit is performed by HDMA without CPU intercept, but in the interrupt mode HDLC transmit operation, CPU write the frame data to HDLC transmit FIFO. HDLC controller has to register to write frame data to HDLC FIFO. The HTXFIFOC, and HTXFIFOT is used for this purpose. When transmit a data, CPU can write the frame data to the HTXFIFOC, TxFIFO frame continue register, and at the end of transmit CPU write last frame data to HTXFIFOT, TxFIFO frame terminate register . the source code for write transmit frame data to FIFO, WriteDataToFifo() is listed in **Listing 3-40**. This function is always called from HDLC interrupt service routine for FIFO available interrupt. The HDLC Tx FIFO write flow is depicted in **Figure 3-23**.

Listing 3-39. SendHdlcFrame() function for KS32C5000(A) (HDLCINIT.C)

```
/*
 *  Function : SendHdlcFrame
 *  Description : Send HDLC frame
 */
int SendHdlcFrame(U32 channel, U8 *Data, int Size)
{
        sBufferDescriptor       *CTxBDPtr ;
        HDLC_Device_Entry       *Dev ;
        U32                     *DataBufferPtr ;
        U8                      *DataBuffer ;

        // Step 1. Get device entry pointer
        Dev = (HDLC_Device_Entry *)&HDLC_Dev[channel] ;

        // Step 2. Get transmit buffer descriptor pointer
        CTxBDPtr = (sBufferDescriptor *)pCTxBDPtr[channel] ;

        // Step 3. Get data buffer pointer
        DataBufferPtr  = (U32 *)CTxBDPtr->BufferDataPtr ;
        DataBuffer  = (U8 *)CTxBDPtr->BufferDataPtr ;

        // Step 4. Check DMA ownership, if setted to 1 then exit send frame
        if ( ((U32)CTxBDPtr->BufferDataPtr & BOwnership_DMA) ) return 0 ;

        // Step 5. Copy Tx frame data to frame buffer
        CTxBDPtr->LengthField = Size ;
        memcpy ((U8 *)DataBuffer,(U8 *)Data,Size);

        // Step 6. Change ownership to DMA
        CTxBDPtr->BufferDataPtr |= BOwnership_DMA;

        // Step 7. Change current buffer descriptor pointer
        pCTxBDPtr[channel] = (U32)CTxBDPtr->NextBufferDescriptor ;

        // Step 8. Set HDLC transmit DMA or Transmit buffer
        if ( Dev->HDLC_Tx_Mode == MODE_DMA ) {
                // Setup HDLC transmit DMA
                HDMA_Tx_init(channel,(U32)DataBufferPtr,Size) ;
                HDMA_Tx_Enable(channel);
        }
        else {
                // Setup transmit buffer and size to transmit
                ModeInt_TxDataBuffer[channel] = (U8 *)DataBuffer ;
                ModeInt_TxDataSize[channel] = Size ;

                // Enable Transmit FIFO avaliable interrupt
                HINTEN(channel) |= TxFAIE ;
        }

        return 1 ;
}
```

Listing 3-40. WriteDataToFifo() function for KS32C5000(A) (HDLCINIT.C)

```
/*
 *  Function : WriteDataToFifo
 *  Description : Write frame data to HDLC Tx FIFO
 *              This function is used only interrupt mode transmit
 */
int WriteDataToFifo(U32 channel)
{
        int i, j ;
        U32 *Data ;
        U8  *data ;

        for (i=0 ; i<4 ; i++) {
                if ( ModeInt_TxDataSize[channel] > 4 )  {
                        // Read data buffer pointer to transmit
                        Data = (U32 *)ModeInt_TxDataBuffer[channel] ;

                        // Write Tx frame data to FIFO
                        HTXFIFOC(channel) = *Data ;

                        // increase data pointer
                        ModeInt_TxDataBuffer[channel] += 4 ;

                        // decrement word
                        ModeInt_TxDataSize[channel] -= 4 ;
                }
                else {
                        // excuted when left transmit byte is 4 or under 4 byte
                        // Disable Transmit FIFO avaliable interrupt
                        HINTEN(channel) &= ~TxFAIE ;

                        if (ModeInt_TxDataSize[channel] == 4 ) {
                                // if lefted transmit data is 1 word, then write
                                // word data to HTXFIFOT register
                                Data = (U32 *)ModeInt_TxDataBuffer[channel] ;
                                HTXFIFOT(channel) = *Data ;
                                ModeInt_TxDataSize[channel] -= 4 ;   // decrement word
                        } else {
                                // Get byte data pointer
                                data = (U8 *)ModeInt_TxDataBuffer[channel] ;
                                HDLCTXFIFOT = (U8 *)&HTXFIFOT(channel) ;
                                HDLCTXFIFOC = (U8 *)&HTXFIFOC(channel) ;

                                // process under 4 byte data
                                for ( j=0 ; j<4; j++) {
                                        ModeInt_TxDataSize[channel] -- ;// decrement byte
                                        if (ModeInt_TxDataSize[channel] > 0)  {
                                                *HDLCTXFIFOC = *data++ ;
                                        }
                                        else {
                                                // end of frame data
```

```
                                        *HDLCTXFIFOT = *data++ ;
                                        break ;
                                    }
                                }
                            }

                            return 0 ; // return end of frame transfer
                        }

                }
                return 1 ; // return not end of frame transfer
}
```

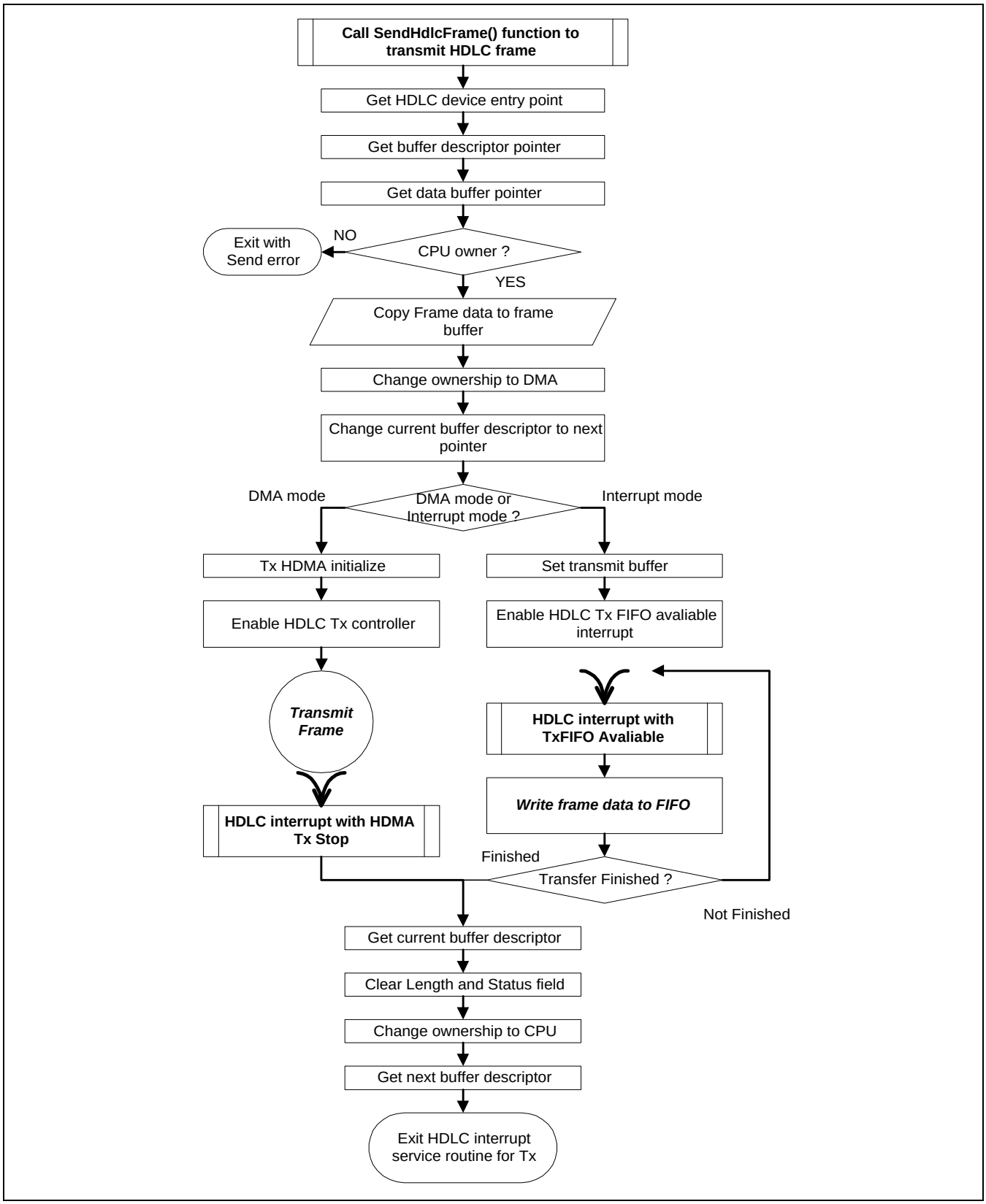


Figure 3-22. HDLC Transmit Frame Data Flow for KS32C5000(A)

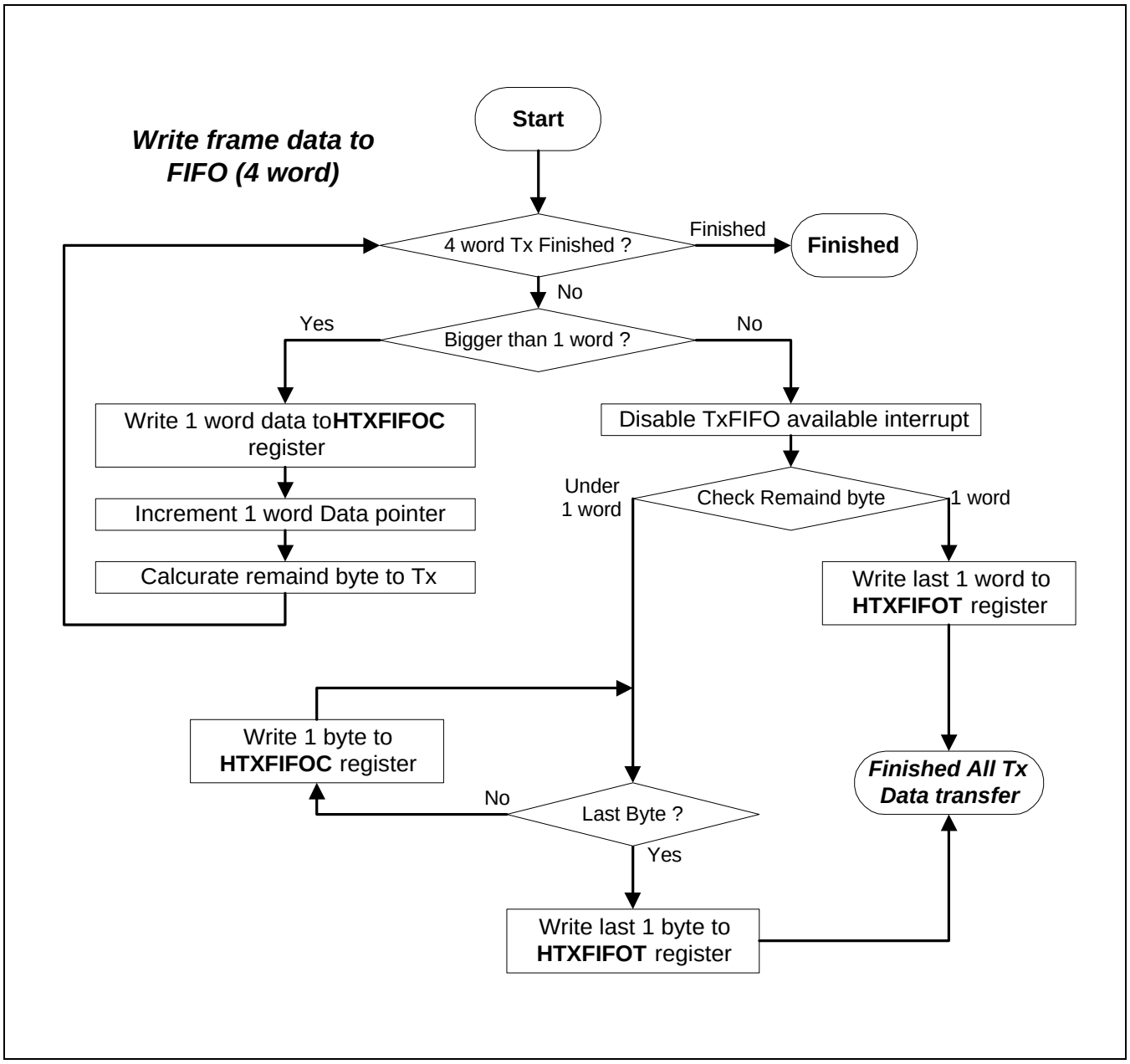


Figure 3-23. HDLC Tx FIFO Write Flow for KS32C5000(A)

Finally, When you transmit HDLC frame, you can follow this step.

STEP 1. Get device entry pointer

STEP 2. Get transmit buffer descriptor pointer

STEP 3. Get data buffer pointer

STEP 4. Check DMA ownership, if setted to 1 then exit send frame

STEP 5. Copy Tx frame data to frame buffer

STEP 6. Change ownership to DMA

STEP 7. Change current buffer descriptor pointer to next pointer

STEP 8. Set HDLC transmit DMA or Transmit buffer

In the DMA mode, all data transfer of HDLC frame is performed by HDMA, but in the Interrupt mode, all data transfer is performed by CPU, when TxFIFO available interrupt is occurred. The interrupt service routine source code for HDLC transmit operation is listed in **Listing 3-41**.

Listing 3-41. Interrupt service routine for HDLC Tx operation with KS32C5000(A) (HDLCINIT.C)

```
.....
        // 6. HDLC Transmit DMA mode Complete
        if ( IntHdlcStatus & (DTxSTOP | DTxABT) ) {
                if ( IntHdlcStatus & DTxSTOP ) {
                        gHdlcTxStatus[channel].DMATxStop++ ;
                        HSTAT(channel) |= DTxSTOP ;
                }
                if ( IntHdlcStatus & DTxABT ) {
                        gHdlcTxStatus[channel].DMATxABT++ ;
                        HSTAT(channel) |= DTxABT ;
                }

                do {
                        // Step 1. Get current transmit buffer descriptor pointer
                        //       and length
                        CTxBDPtr = (sBufferDescriptor *)gCTxBDPtr[channel];
                        TxLength = CTxBDPtr->LengthField ;

                        // Step 2. Clear Length and Status field
                        CTxBDPtr->LengthField = (U32)0x0;
                        CTxBDPtr->StatusField = (U32)0x0;

                        // Step 3. Get Next buffer descriptor
                        gCTxBDPtr[channel] = (U32)CTxBDPtr->NextBufferDescriptor ;

                        // Step 4. Clear DMA owner to CPU
                        CTxBDPtr->BufferDataPtr &= BOwnership_CPU ;

                } while (pCTxBDPtr[channel] == gCTxBDPtr[channel]) ;
        }

        // 7. HDLC Transmit Interrupt mode
```

```
        if (Dev->HDLC_Tx_Mode == MODE_INTERRUPT) {
                if  (IntHdlcStatus & TxFA) {
                        gHdlcTxStatus[channel].TxFIFOAvailable++ ;

                        if ( ModeInt_TxDataSize[channel] > 0)
                                // Step 1. write 4 word frame data to FIFO
                                if ( !WriteDataToFifo(channel) ) {

                                        // Step 2. Get current receive buffer descriptor point
                                        CTxBDPtr = (sBufferDescriptor *)gCTxBDPtr[channel];
                                        TxLength = CTxBDPtr->LengthField ;

                                        // Step 3. Clear length and status field
                                        CTxBDPtr->LengthField = (U32)0x0;
                                        CTxBDPtr->StatusField = (U32)0x0;

                                        // Step 4. Get Next buffer descriptor
                                        gCTxBDPtr[channel] =
                                                (U32)CTxBDPtr->NextBufferDescriptor ;

                                        // Step 5. Clear owner to CPU
                                        CTxBDPtr->BufferDataPtr &= BOwnership_CPU ;
                                }
                }

        }

        // 8. HDLC Transmit Status
        if ( IntHdlcStatus & ( TxFC | TxU | TxCTS | TxSCTS ) )    {
                if ( IntHdlcStatus & TxFC ) {
                        gHdlcTxStatus[channel].TxFrameComplete++ ;
                        HSTAT(channel) |= TxFC ;
                }
                if ( IntHdlcStatus & TxU ) {
                        gHdlcTxStatus[channel].TxUnderrun++ ;
                        HSTAT(channel) |= TxU ;
                }
                if (IntHdlcStatus & TxCTS)
                        gHdlcTxStatus[channel].TxLevelOfCTS++ ;

                if (IntHdlcStatus & TxSCTS) {
                        gHdlcTxStatus[channel].TxTransitionOfCTS++ ;
                        }
        }
```

Receive HDLC frame with KS32C5000(A)

After setting all control register and HDLC receive buffer descriptor, you can receive HDLC frame, At the first time, receive HDLC buffer descriptor has DMA owner, after receive, in the interrupt service routine, CPU change this owner bit to CPU, then CPU can use this buffer descriptor and data buffer to process. After process the data on this buffer descriptor, CPU should change the owner to DMA.

As same as HDLC frame data transfer operation, HDLC receive operation can be used with DMA, and interrupt mode. In the DMA mode, HDLC controller can receive incoming frame without CPU intervention, but in the interrupt mode operation, CPU should read the received data from HDLC RxFIFO.
The source code for interrupt service routine is listed in **Listing 3-42**.

Listing 3-42. Interrupt service routine for HDLC Rx operation with KS32C5000(A) (HDLCINIT.C)

```
.........

        // 3. Process HDMA receive operation
        //    This routine is used when HDLC DMA mode receive operation
        if ( IntHdlcStatus & ( DRxSTOP | DRxABT) ) {

                if ( IntHdlcStatus & DRxSTOP ) {
                        gHdlcRxStatus[channel].DMARxStop++ ;
                        HSTAT(channel) |= DRxSTOP ;
                }
                else if ( IntHdlcStatus & DRxABT ) {
                        gHdlcRxStatus[channel].DMARxABT++ ;
                        HSTAT(channel) |= DRxABT ;
                }

                // Step 1. Get current receive buffer descriptor point
                CRxBDPtr = (sBufferDescriptor *)gCRxBDPtr[channel];

                // Step 2. Clear owner to CPU
                CRxBDPtr->BufferDataPtr &= BOwnership_CPU ;

                // Step 3. Get length and status
                CRxBDPtr->LengthField = RxLength = HDMARXBCNT(channel) ;
                CRxBDPtr->StatusField = IntHdlcStatus ;

                // Step 4. Get Next buffer descriptor
                gCRxBDPtr[channel] = (U32)CRxBDPtr->NextBufferDescriptor ;

                // Step 5. Initialize HDLC DMA for receive
                HDLC_Rx_init(channel) ;
        }


        // 4. Process HDLC receive operation
        //    This routine is used when HDLC interrupt mode receive operation
        if (Dev->HDLC_Rx_Mode == MODE_INTERRUPT) {
                if (IntHdlcStatus & (RxFA | RxFV | RxFERR) ) {
                        if (IntHdlcStatus & RxFA)
```

```
                        gHdlcRxStatus[channel].RxFIFOAvalable++ ;

                if (IntHdlcStatus & (RxFV | RxFERR) ) {

                        if (IntHdlcStatus & RxFV)
                                gHdlcRxStatus[channel].RxLastFrameValid++ ;

                        if ( IntHdlcStatus & RxFERR )
                                gHdlcRxStatus[channel].RxFrameError++ ;

                        // Step 1. Read received data from HDLC receive FIFO entry
                        ReadDataFromFifo(channel,(HSTAT(channel)&0xF)) ;

                        // Step 2. Get current receive buffer descriptor point
                        CRxBDPtr = (sBufferDescriptor *)gCRxBDPtr[channel];

                        // Step 3. Clear owner to CPU
                        CRxBDPtr->BufferDataPtr &= BOwnership_CPU ;

                        // Step 4. Get length and status
                        CRxBDPtr->LengthField = RxLength =
                                        ModeInt_RxDataSize[channel] ;
                        CRxBDPtr->StatusField = IntHdlcStatus ;

                        // Step 5. Get Next buffer descriptor
                        gCRxBDPtr[channel] = (U32)CRxBDPtr->NextBufferDescriptor ;

                        // Step 6. Initialize HDLC DMA for receive
                        HDLC_Rx_init(channel) ;
                } else
                        // Read received data from HDLC receive FIFO entry
                        ReadDataFromFifo(channel, 15) ;
        }
}

// 5. Save HDLC receive status
if ( IntHdlcStatus & ( RxOV|RxABT|RxAERR|RxFAP|RxFD|RxDCD|RxSDCD|RxIDLE) ) {

        // HDLC Rx OverRun Error
        if ( IntHdlcStatus & RxOV ) {
                gHdlcRxStatus[channel].RxOverrun++ ;
                HSTAT(channel) |= RxOV ;

                // check all descriptor is used, then enable HDMA Rx
                if ( gCRxBDPtr[channel] == pCRxBDPtr[channel] )
                        HDLC_Rx_init(channel) ;
        }

        if ( IntHdlcStatus & RxABT ) {
                gHdlcRxStatus[channel].RxAbort++ ;
                HSTAT(channel) |= RxABT ;
                HDLC_Rx_init(channel) ;
        }
```

```
                if ( IntHdlcStatus & RxAERR )
                        gHdlcRxStatus[channel].RxAddressError++ ;

                if (IntHdlcStatus & RxFAP)
                        gHdlcRxStatus[channel].RxFIFOAddrPresent++ ;

                if (IntHdlcStatus & RxFD) {
                        gHdlcRxStatus[channel].RxFlagDetected++ ;
                        HSTAT(channel) |= RxFD ;
                        }
                if (IntHdlcStatus & RxDCD)
                        gHdlcRxStatus[channel].RxLevelOfDCD++ ;

                if (IntHdlcStatus & RxSDCD) {
                        gHdlcRxStatus[channel].RxTransitionOfDCD++ ;
                        HSTAT(channel) |= RxSDCD ;
                        }
                if (IntHdlcStatus & RxIDLE) {
                        gHdlcRxStatus[channel].RxIdle++ ;
                        HSTAT(channel) |= RxIDLE ;
                        }
        }
.........
```

In the interrupt service routine for receive operation, store the received frame data, receive frame length, and status. So user program that process received frame, can handle this received frame with buffer descriptor information. The sample code for processing received frame is listed in **Listing 3-43**.

Listing 3-43. ReceiveHdlcFrame() function for KS32C5000(A) (HDLCINIT.C)

```
/*
 *  Function : ReceiveHdlcFrame
 *  Description : Receive HDLC frame
 */
int ReceiveHdlcFrame(U32 channel)
{
        sBufferDescriptor       *pRxBDPtr ;
        U32                     CRxBDPtr ;
        U32                     *DataBuffer ;
        U32                     Length ;
        U32                     Status ;

        // Step 1. Get current frame buffer pointer
        CRxBDPtr = (U32)gCRxBDPtr[channel] ;

        do {
                // Step 2. Get frame buffer pointer for process
                pRxBDPtr = (sBufferDescriptor *)pCRxBDPtr[channel] ;

                // Step 3. Check Ownership is CPU or not
```

```
                if ( !((U32)(pRxBDPtr->BufferDataPtr) & BOwnership_DMA) )  {

                        // Step 4. If ownership is CPU, then receive frame is exist
                        //        So, get this frame to process
                        DataBuffer = (U32 *)pRxBDPtr->BufferDataPtr ;
                        Length = pRxBDPtr->LengthField  ;
                        Status = pRxBDPtr->StatusField  ;

                        if ( !( Status & (RxFERR | DRxABT) ) ) {
                                memcpy ((U8 *)&TempBuf,(U8 *)DataBuffer,Length);
                        }
                }
                else break ;

                // Step 5. Change owner to DMA
                (pRxBDPtr->BufferDataPtr) |= BOwnership_DMA;
                pRxBDPtr->LengthField = (U32)0x0;
                pRxBDPtr->StatusField = (U32)0x0;

                // Step 6. check all descriptor is used, then enable HDLC receive
                if ( gCRxBDPtr[channel] == pCRxBDPtr[channel] )
                        HDLC_Rx_init(channel) ;

                // Step 7. Get Next Frame Descriptor pointer to process
                pCRxBDPtr[channel] = (U32)(pRxBDPtr->NextBufferDescriptor) ;

        } while (CRxBDPtr != pCRxBDPtr[channel]);

        return 1 ;
}
```

When you processing the received frame, you can follow this step.

STEP 1. Get current frame buffer pointer

STEP 2. Get frame buffer pointer for process

STEP 3. Check Ownership is CPU or not

In this step, if the owner is DMA, then this buffer can' t be used. So at the first time, you should check this owner bit.

STEP 4. Get the frame to process

In this step, you can check received frame by status field in the buffer descriptor, if there is no receive error, then you can get received frame, and process it, but if have any error, then just leave out from receive operation.

STEP 5. Change owner to DMA

After process received frame, CPU should change owner bit to DMA, if this bit is not changed, then next HDLC initialize routine can' t use this frame buffer.

STEP 6. Check all descriptor is used, then enable HDLC receive

If buffer descriptor is already full, then enable HDLC controller and HDMA controller enable to receive incoming frame data.

STEP 7. Get Next Frame Descriptor pointer to process

The flow of receive and processing the received data is depicted in **Figure 3-24**, and **Figure 3-25**.

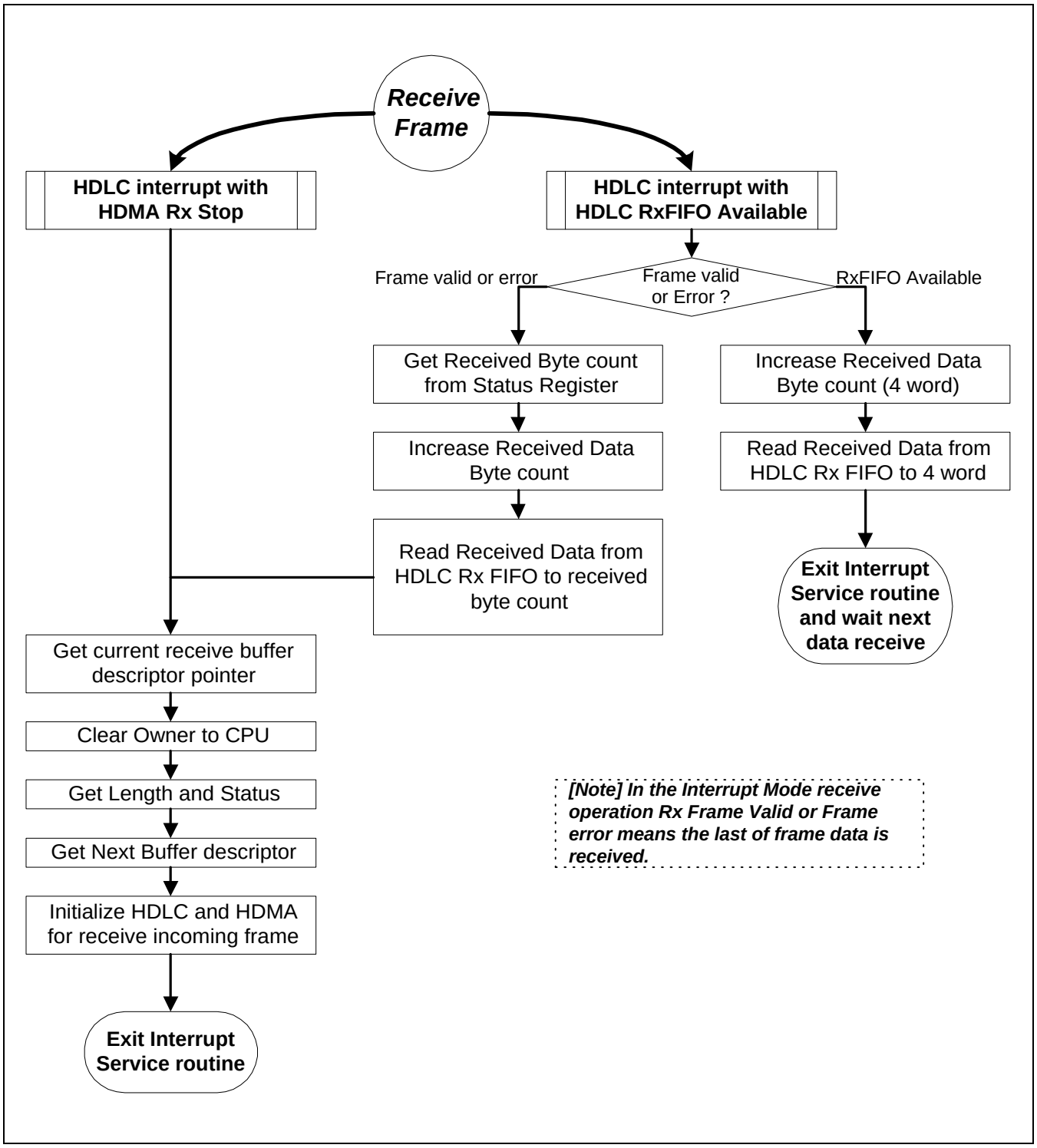


Figure 3-24. HDLC Receive Flow for KS32C5000(A)

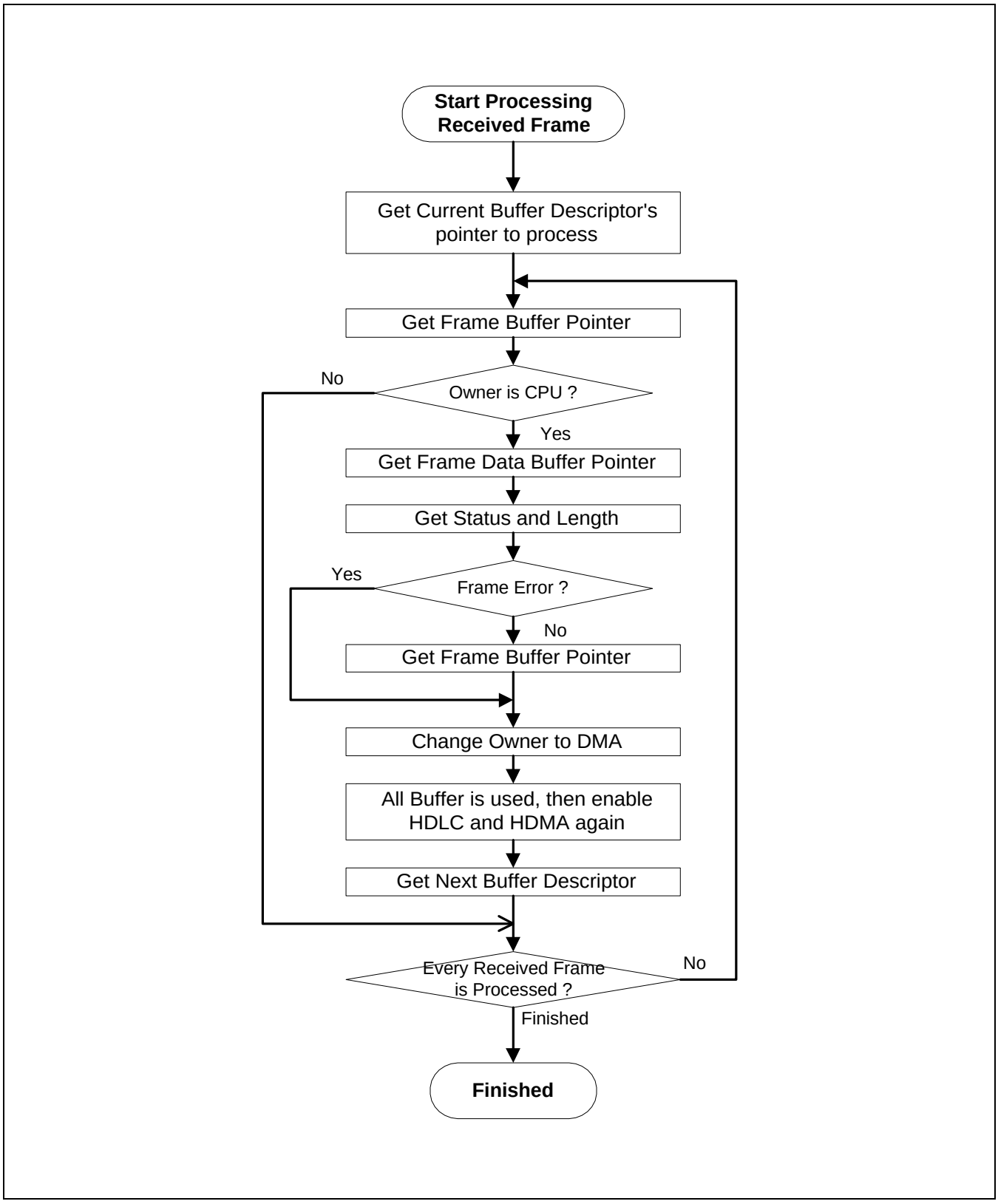


Figure 3-25. HDLC Processing Received Frame Flow for KS32C5000(A)

HIGH-LEVEL DATA LINK CONTROLLER FOR KS32C50100

HDLC Diagnostic Code Function

The diagnostic source code for the HDLC (High-Level Data Link Controller) is composed of four files, HDLC100.H, HDLC100.C, HDLCINIT100.C, and HDLCLIB100.C.

HDLC100.H: Definition file for HDLC diagnostic code, Register bit value, frame structure, frame descriptor structure, and function prototype.

HDLCMAIN.C: The main diagnostic code function call for HDLC function test.

HDLCINIT100.C: Initialize HDLC and HDMA controller for normal operating environment, each interrupt service routine.

HDLCLIB100.C: The library functions for diagnostic code.

SNDS.H

Listing 3-46. System Control Registers (SNDS.H)

```
#define Vprint   *(volatile unsigned int *)
#define Base_Addr                                0x3ff0000

// System Manager Register
#define SYSCFG                                   (Vprint(Base_Addr+0x0000)
                :

// GDMA 1
                :
```

Hardware control registers used in the diagnostic C-source code are defined in SNDS.H header file. In this header file , Base_Addr has the same value as the base address configured at boot code on the reset time. The address of system configuration register is typically calculated by adding the register' s offset to Base_Addr. The base address (reset value = 0x3ff0000) can be changed by writing a new base address to the special register bank base pointer field of the SYSCFG register. Base_Addr should be updated before compiling and linking the diagnostic code. A control register should be declared as volatile in order to avoid certain compiler optimizations that prevent the value of variable from being accessed correctly as the expression indicates. For example :

```
volatile int clock;
int timer1;
timer1 = clock;
if(timer1 == clock) ...;
```

If the clock is not declared as volatile , some compilers can optimize the two expressions in such a way that the value of the clock is examined only once.

Definitions for HDLC

Listing 3-47. New Definition for HDLC (HDLC100.H)

```
 // HDLC Registers
#define HMODE(channel)                  (VPint(Base_Addr+0x7000 + channel*0x1000))
#define HCON(channel)                   (VPint(Base_Addr+0x7004 + channel*0x1000))
#define HSTAT(channel)                  (VPint(Base_Addr+0x7008 + channel*0x1000))
#define HINTEN(channel)                 (VPint(Base_Addr+0x700c + channel*0x1000))
#define HTXFIFOC(channel)               (VPint(Base_Addr+0x7010 + channel*0x1000))
#define HTXFIFOT(channel)               (VPint(Base_Addr+0x7014 + channel*0x1000))
#define HRXFIFO(channel)                (VPint(Base_Addr+0x7018 + channel*0x1000))
#define HBRGTC(channel)                 (VPint(Base_Addr+0x701c + channel*0x1000))
#define HPRMB(channel)                  (VPint(Base_Addr+0x7020 + channel*0x1000))
#define HSAR0(channel)                  (VPint(Base_Addr+0x7024 + channel*0x1000))
#define HSAR1(channel)                  (VPint(Base_Addr+0x7028 + channel*0x1000))
#define HSAR2(channel)                  (VPint(Base_Addr+0x702c + channel*0x1000))
#define HSAR3(channel)                  (VPint(Base_Addr+0x7030 + channel*0x1000))
#define HMASK(channel)                  (VPint(Base_Addr+0x7034 + channel*0x1000))
#define HDMATXPTR(channel)              (VPint(Base_Addr+0x7038 + channel*0x1000))
#define HDMARXPTR(channel)              (VPint(Base_Addr+0x703c + channel*0x1000))
#define HMFLR(channel)                  (VPint(Base_Addr+0x7040 + channel*0x1000))
#define HRBSR(channel)                  (VPint(Base_Addr+0x7044 + channel*0x1000))

// Register Bit Control Macros
#define HdlcReset(channel)              HCON(channel) |= TxRS|RxRS|DTxRS|DRxRS ;
#define HDLCTxEN(channel)               HCON(channel) |= TxEN
#define HDLCRxEN(channel)               HCON(channel) |= RxEN
#define HDMATxEN(channel)               HCON(channel) |= DTxEN
#define HDMARxEN(channel)               HCON(channel) |= DRxEN
#define HDLCLoopbackEn(channel)         HCON(channel) |= TxLOOP
#define HDLCLoopbackDis(channel)        HCON(channel) &= ~TxLOOP
#define DRxSizeSet(channel,size)        HRBSR(channel) = size
#define MaxFrameLengthSet(channel,size)         HMFLR(channel) = size
#define InterCLKEN(channel)             HCON(channel) |= DPLLEN|BRGEN ;
#define HDLCTxGo(channel)               HDLCTxEN(channel); HDMATxEN(channel)
#define HdlcTxReset(channel)            HCON(channel) |= TxRS
#define HdlcRxReset(channel)            HCON(channel) |= RxRS ;
#define HdlcDMATxReset(channel)         HCON(channel) |= DTxRS ;
#define HdlcDMARxReset(channel)         HCON(channel) |= DRxRS ;
#define ShowChan(channel)               ch=(!channel)?'A':'B'
```

HDLC has two operating mode which are HDMA and Interrupt. Each mode should have the different register setting and the special interrupt service routine. From this after , we will describe on HDMA mode operation without special comment. For Interrupt mode , we spare the last paragraph of this section and the paragraph of **The Way to Use Status Register.**

HDLC Initialize

The HDLC and HDMA should be initialized before getting into operation. HdlcInitialize() function is programmed to accomplish this initialization. The following sections describe the contents of this function.

The KS32C50100 has HDLC and HDMA controller for High-Level Data Link interface. The HDMA is used for transferring and receiving data to memory and transfer the transmit data to HDLC. The HDLC can support up to 10Mbps and Full duplex operation using an external/internal clock. So you need to set the HDLC, and HDMA controller to work properly.

The detail of HDLC initialize function is described in source code **listing 3-48** and **figure 3-26**.

Listing 3-48. HdlcInitialize() function (HDLC100INIT.C)

```
/*
 * Function       : HdlcInitialize
 * Description    : This function initialize the HDLC Block
 */
void HdlcInitialize (void)
{
        U8      channel;        U32     UserArea;

        for(channel=HDLCA;channel<=HDLCB;channel++)
        {
                //Reset All HDLC block
                HdlcReset(channel) ;

                // Internal Register Setting for Tx/Rx operation
                HMODE(channel)          = gMode  = NRZ | TxCBO2 | RxCRxC | BRGMCLK | TxOTxC ;
                HCON(channel)           = gControl =Tx1WD | Rx1WD | DTxSTSK | DRxSTSK | RxWA0 |
TxDTR | BRGEN;
                HINTEN(channel)   = gHIntEN = DTxFDIE | DRxFDIE | DRxNLIE | DRxNOIE | RxMOVIE;
                HBRGTC(channel)  = GetBaudRate(HDLCBaud) ;
                AddressSet(channel) ;
                MaxFrameLengthSet(channel,MaxFrameData);
                HRBSR(channel)          = RxBufLength ;

                // Initialize Buffer Descriptor
                TxBD_init(channel) ;
                RxBD_init(channel) ;
                TxDataWrite(channel) ;

                // Ready to receive Data
                HDMARxEN(channel) ;
                HDLCRxEN(channel) ;

        }

        // Interrupt Vector Setup
        SysSetInterrupt(nHDLCTxA_INT, HDLCTxA_isr );
        SysSetInterrupt(nHDLCRxA_INT, HDLCRxA_isr );
        SysSetInterrupt(nHDLCTxB_INT, HDLCTxB_isr );
        SysSetInterrupt(nHDLCRxB_INT, HDLCRxB_isr );
```

```
        Enable_Int(nHDLCTxA_INT);
        Enable_Int(nHDLCRxA_INT);
        Enable_Int(nHDLCTxB_INT);
        Enable_Int(nHDLCRxB_INT);
        Enable_Int(nGLOBAL_INT);


        // Initialize Global Variables

}
```

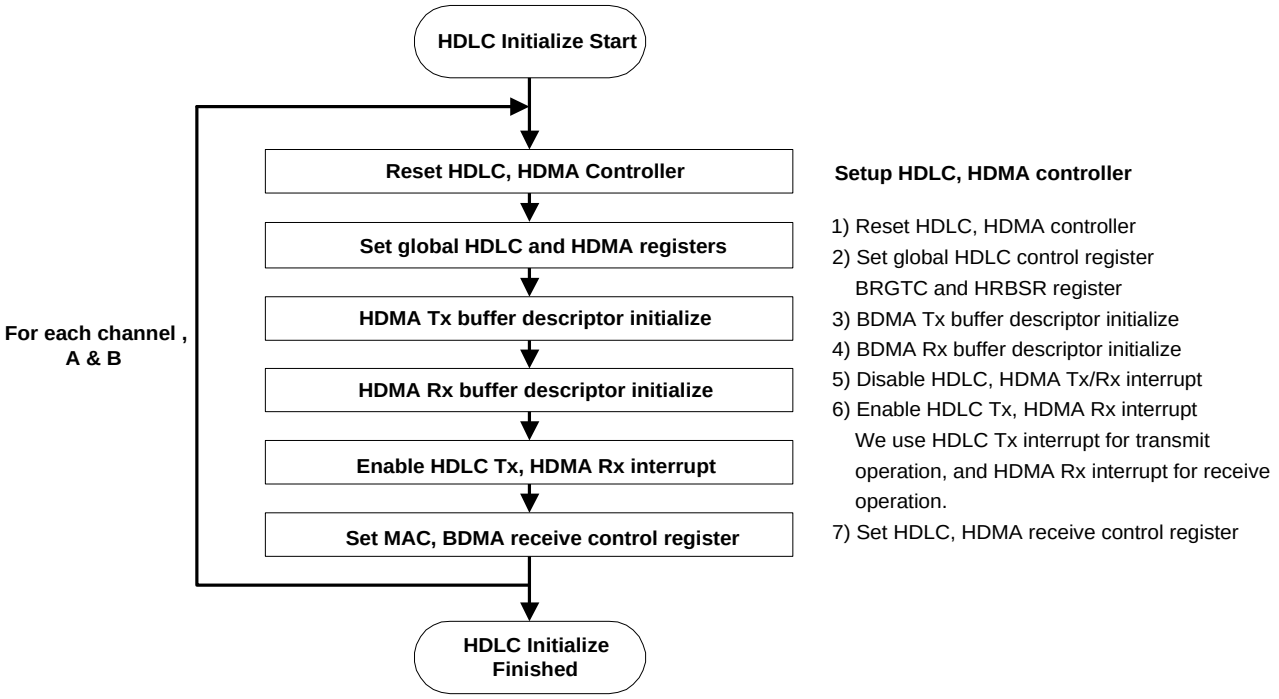


Figure 3-26. HDLC Initialize flow

The each step of HDLC initializing function is described as followings.

STEP 1 : Resettig HDLC and HDMA block

First of all , it is necessary to reset HDLC and DMA controller , which initialize registers as defaults values. See **Listing 3-47** and **Listing 3-48.**

```
for(channel=HDLCA;channel<=HDLCB;channel++)
{
        //Reset All HDLC block
        HdlcReset(channel) ;
}
```

STEP 2 : Setting initial condition of HDMA and HDLC

STEP 2.1 :Selecting the clock mode and clock selection

Register	Bit Name	Description	Example	
			Control Bit	Behavior
HMODE	RxCLK	**Selecting Rx clock** Rx clock can be selected one of output clock from BRG or DPLLOUTR or external TxC/RxC pin.	RxCRxC	External clock to RxC pin is selected as Rx clock.
	TxCLK	**Selecting Tx clock** Tx clock can be selected one of output clock from BRG or DPLLOUTT or external TxC/RxC pin.	TxCBO2	BRGOUT2 clock generated by BRG block is selected as Tx clock.
	BRGCLK	**Internal BRG clock for Tx/Rx clock** BRG source clock can be select between MCLK and RxC pin. If you use a clock derived from BRG block as Tx/Rx clock of HDLC, enable BRGEN bit in HCON register. **NOTE:** Using register HBRGTC, the frequency of Baud Rate Clock is determined (See STEP 2.1.1 Baud Rate Generator).	BRGMCLK	System Main clock selected as BRG source clock.
	DPLLCLK	**Internal DPLL clock for Tx/Rx clock** DPLL source clock can be selected among TxC/RxC pin, MCLK and BRGOUT1, 2 .If you use a clock derived from DPLL block as Tx/Rx clock of HDLC, enable DPLLEN bit in HCON register.	–	Not used in diagnostic program.
HCON	BRGEN	**BRG enable** Enable BRG counter and Start generating BRGOUT1,2	BRGEN	BRG counter loads BRGTC and counting down for generating BRGOUT1,2
	DPLLEN	DPLL enable Enable DPLL and start finding a locking edge in the incoming data.	–	Not used in diagnostic program.

Register	Bit Name	Description	Example	
			Control Bit	Behavior
HBRGTC	CNT0 CNT1 CNT2	See User' s Manual	HBRGTC = 0xXXXX;	BRG counter load this value.
			HBRGTCA = GetBaudRate (HDLCBaud);	GetBaudRate is a function to get BRG counter value. Simply you pass a value of BAUDRATE as parameter .
HMODE	TXCOPS	Selecting TxC pin as Output When TxC pin is not the input clock ,you can monitor internal clocks through TxC pin.	TxOTxC	Tx clock is appeared at TXC pin.

STEP 2.1.1 Baud Rate Generator

You have to set the programmable Baud Rate Generator(BRG) to get a desired baud rate. This HBRGTC register contains a 16-bit time constant register, which is made of a 12-bit down counter for time constant value and two control bits to divide by 16 or 32.

Using the following formula, you can obtain the desired baud rate.

BRGOUT1 = (MCLK2 or RxC) / (CNT0 +1) / (16^{CNT1})
BRGOUT2 = BRGOUT1 / (1 or 16 or 32 according to CNT2 value of the HBRGTC)

By changing CNT2 HBRGTC[1:0] in HBRGTC register , two different BRG can be obtained. The GetBaudRate() program can get BRGTC value as shown in **listing 3-48**. This function get the wanted baud rate as close as possible by using internal 50MHz MCLK and is only for BRGOUT2 and has no consideration for BRGOUT1. So if you want to use external clock to get Baud Rate, a slight modification will be needed like changing the defined MCLK2 value.

Listing 3-48. GetBaudRate(U32 WantedBR) function (HDLC100LIB.C)

```
#define  MCLK2  25000000

/*
 * Function : GetBaudRate
 * Description : HDLC Internal Time Constant Value (BRGTC)
 */
U32 GetBaudRate(U32 WantedBR)
{
        U32     CNT0,SelCNT0,CNT1,CNT2;
        U32     i,scal1,scal2;
        float   CalcedBR,CalcedBR1 ;
        float   DiffVal1 , DiffVal2 , SelDiffVal ;

        SelDiffVal = (float) WantedBR;

        for(i=0;I<6;i++)
```

```
        {
                switch(i) {
                        case 0 : scal1 =  1 ; scal2 =  1 ; break;
                        case 1 : scal1 =  1 ; scal2 = 16 ; break;
                        case 2 : scal1 =  1 ; scal2 = 32 ; break;
                        case 3 : scal1 = 16 ; scal2 =  1 ; break;
                        case 4 : scal1 = 16 ; scal2 = 16 ; break;
                        case 5 : scal1 = 16 ; scal2 = 32 ; break;
                        default : ;
                }
                CNT0 = MCLK2 / WantedBR / scal1 / scal2 ;
                if(CNT0==0 | CNT0 > 4096) continue ;
                CalcedBR = (float) MCLK2 / (float)CNT0++ / (float)scal1 / (float)scal2;
                DiffVal1 = CalcedBR-(float)WantedBR ;
                CalcedBR1 = (float) MCLK2 / (float)CNT0 / (float)scal1 / (float)scal2;
                DiffVal2 = (float)WantedBR-CalcedBR1 ;

                if(DiffVal1 > DiffVal2) {
                        DiffVal1 = DiffVal2 ;
                        CalcedBR = CalcedBR1 ;
                        CNT0 -- ;
                } else CNT0-=2 ;

                if(DiffVal1<SelDiffVal){
                        SelCNT0 = CNT0 ;
                        CNT1 = scal1 / 16 ;
                        CNT2 = scal2 / 16 ;
                        SelDiffVal = DiffVal1 ;
                }
        }
        return ((SelCNT0 <<4) | (CNT1<<2) | CNT2 );
}
```

STEP 2.1.2 Using DPLL Clock

If you want to correct RxC /TxC phase with DPLL, DPLL clock can be used. The most important thing you should obey is that the selected DPLL clock is faster than the data rate of RxD as following table. DPLLOUTT for Tx clock and DPLLOUTR for Rx clock can be selected respectively.

Data Type	Clock Rate
NRZ/NRZI	32 times faster than RxD
FM/Manchester	16 times faster than RxD

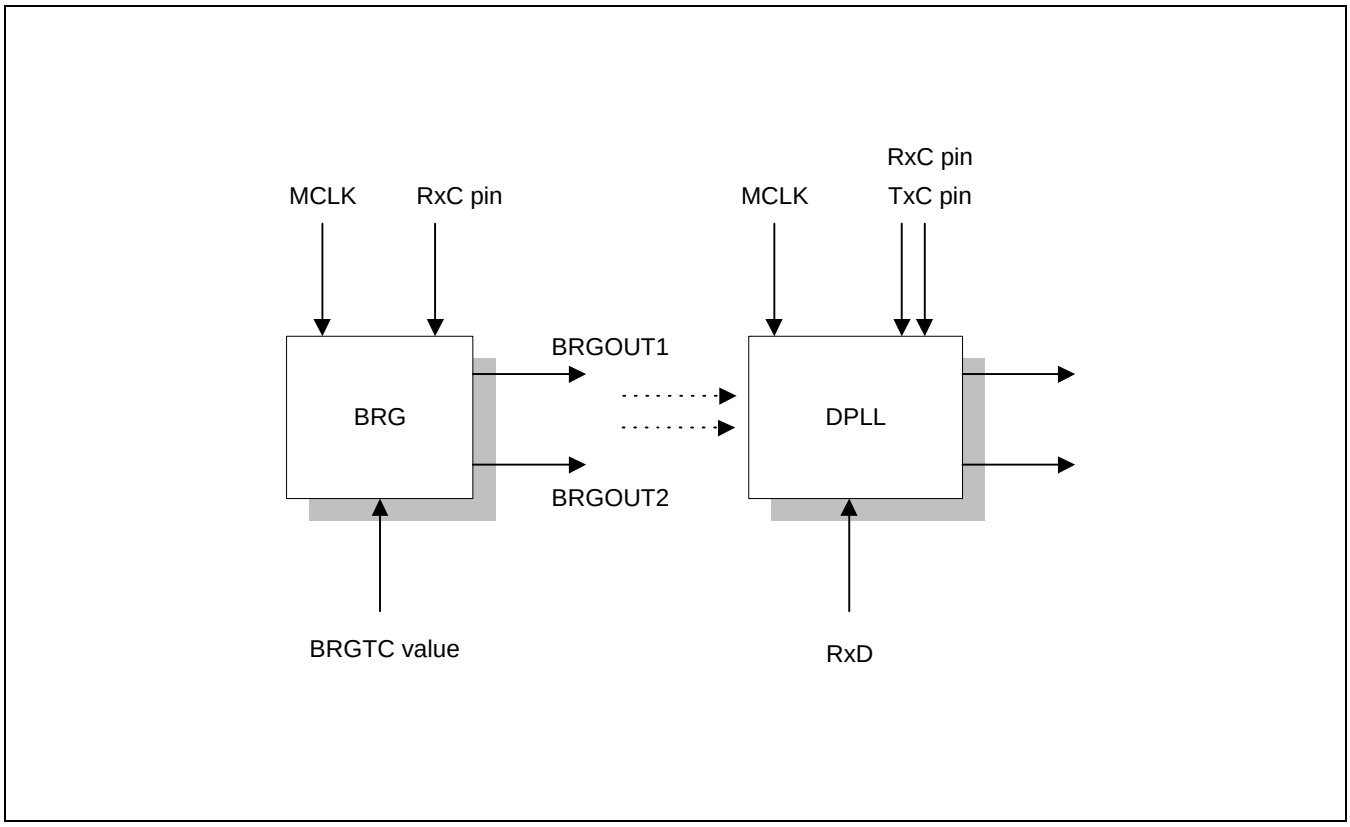


Figure 3-27. HDLC Internal clock Block Diagram

STEP 2.2 Selecting Data Format/Flag Pattern/Endian Mode

Register	Bit Name	Description	Example	
			Control Bit	**Behavior**
HMODE	DF	**Selecting Data Format** You can use NRZ/NRZI and FM0/FM1/Manchester data format provided by KS32C50100 .	NRZ	NRZ data format will be used for Data Encoding/Decoding for channel A.
	TxLittle	**Selecting Endian mode** It is depends on TxLittle and RxLittle bits whether data bytes are swapped or not between system bus and HDLC Tx/Rx FIFO. Data bytes are swapped in default mode when you set Little pin LOW and TxLittle and RxLittle is disabled.(i.e. Normal Big Endian mode) The reason is why Tx/Rx FIFO is Little Endian mode.	(Default)	The transmitted data will be a Big Endian format.
	RxLittle		(Default)	We assume the data which will be received as Big Endian format.
	TXPL	**Selecting Preamble Length** You can select the length of preamble to be sent before opening flag.	–	Not used in diagnostic program.
HCON	TxPRMB	**Selecting Flag pattern** When this bit is set to '1' ,your pattern will be transmitted. And this bit must be disabled after Tx enabled. If not, the data to be transmitted can not be transmitted. In case TxPRMB bit is ‘ 0’ , flag or mark idle is selected.	–	Not used in diagnostic program.
HPRMB	Preamble Pattern [7:0]	**Setting Special Idle Pattern** If you want to transmit special idle pattern, you must write this into HPRMB (8-bit) register.	–	Not used in diagnostic program.

STEP 2.3 Setting the Interrupt Enable bits

You can manage HDLC with best performance, when the interrupt bits are set correctly and interrupt service routines are programmed concisely for your system. These bits can be enabled when TxEN,RxEN,DTxEN and DRxEN are set in HCON register. See **Listing 3-47** , **Listing 3-48** and **STEP 6**

```
for(channel=HDLCA;channel<=HDLCB;channel++)
{
        // Setting Interrupt Enable
        HINTEN(channel) = DTxFDIE | DRxFDIE | DRxNLIE | DRxNOIE | RxMOVIE ;
        HDMATxEN(channel);   HDLCTxEN (channel);
}
```

After this statement, HDLC and HDMA interrupt can be used.

STEP 2.4 Setting Station Address

There are four station address registers and one mask register to recognize address(HSADR0~3 and HMASK) Each registers consist of 32-bits. The value of bits in HMASK represents whether the consistent bits of HSADR0~3 will be compared with the incoming address field. If the address is not matched, the corresponding frame is discarded. **Listing 3-50** show that if the first 4-bytes of HDLC incoming bit steam is one of 0x1234, 0xabcd, 0xffff and 0xaaaa, it will be accepted as valid address.

Listing 3-50. AddressSet(U8 Channel) function (HDLC100LIB.C)

```
/*
 * Function : AddressSet
 * Description : HDLC Station address
 */
void AddressSet(U8 channel)
{
        HSAR0(channel) = 0x12345678 ;
        HSAR1(channel) = 0xabcdef01 ;
        HSAR2(channel) = 0xffffffff ;
        HSAR3(channel) = 0xaaaaaaaa ;
        HMASK(channel) = 0xfffff0000;
}
```

STEP 2.5 Setting Other Registers for DMA operation

There are two 16-bit-long registers which should be set before DMA operation. One is HMFLR limiting the length of incoming frame data, and the other is HRBSR determining the buffer size of a Rx Buffer Descriptor. If the frame data received exceeds HMFLR register value, the frame is discarded and FLV(Frame Length Violation) bit is set in the corresponding buffer descriptor. See **Listing 3-47** and **Listing 3-48.**

```
for(channel=HDLCA;channel<=HDLCB;channel++)
{
        //Reset All HDLC block
        MaxFrameLengthSet(channel,MaxFrameLength) ;
        HRBSR(channel)          = RxBufLength ;
}
```

STEP 3 Initializing the HDMA Tx/Rx Frame Descriptor

HDLC uses HDMA to receive and transmit frame data. And HDMA use a buffer descriptor structure to interface with memory automatically. The transmit HDMA frame descriptor has frame buffer pointer, control field, length and status field, and next frame descriptor field, the HDMA receive frame descriptor is almost same as transmit frame descriptor except control field.

The basic frame descriptor structure is described in Listing 3-51, and Figure 3-27.

Listing 3-51. Buffer Descriptor Structure (HDLC100.H)

```
// Tx/Rx Buffer descriptor structure (With Previous Descriptor)
typedef struct BD {
        U32 BufferDataPtr;
        U32 Reserved;           // cf: RX-reserved, TX-Reserved(25bits) + Control bits(7bits)
        U32 StatusLength;
        struct BD *NextBD ;
        struct BD *PrevBD ;
} sBufferDescriptor;
```

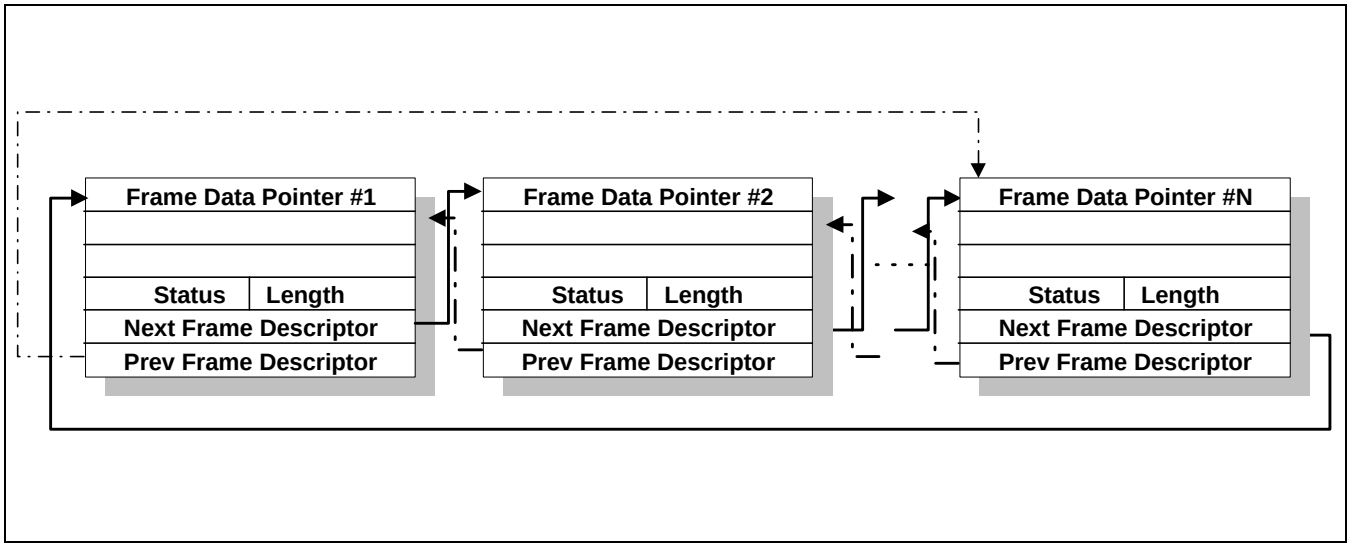


Figure 3-28. HDMA Buffer Descriptor Structure

The HDMA Tx/Rx frame descriptor, and frame buffer area should be non-cacheable, because HDMA can update the value, so when we initialize buffer descriptor , oring NonCache(= 0x4000000) for non-cacheable access. The **Listing 3-52**, and **Listing 3-53** show the detail operation of setup HDMA Tx/Rx frame descriptor.
The default owner of transmit HDMA frame descriptor is CPU, and the default owner of receive HDMA owner is HDMA, after receive frame, HDMA controller change the owner bit on HDMA frame descriptor to CPU owner , then it can be used by CPU.

Listing 3-52. TxBD_init(U8 channel) function (HDLC100INIT.C)

```
/*
 * void TxBD_init(U8 channel) ;
 *          Initialize Tx buffer descriptor
 */
void TxBD_init(U8 channel)
{
        sBufferDescriptor       *psTxBD ;
        sBufferDescriptor       *psStartBD ;
        sBufferDescriptor       *psPrevBD ;
        U32                             TxBDataStartAddr ;
        U32                             TxBufLength ;
        U32                             i ;
        U32                             MisAlignBytes ;

        gCTxBDStart[channel] = (U32)sTxBDStart[channel] | NonCache ;
        HDMATXPTR(channel)   =  gCTxBDStart[channel] ;

        psTxBD                          = (sBufferDescriptor *)gCTxBDStart[channel] ;
        psStartBD                       = psTxBD ;
        psTxBD->PrevBD                  = (sBufferDescriptor *)NULL ;

        TxBDataStartAddr        = (U32)TxBBaseAddr ;
        TxBDataStartAddr    |= NonCache ;

        for(i=1; i<=MaxTxBDCount; i++){
                TxBufLength = PatternGen(i) & 0x3ff; //debug
                if(TxBufLength<6)       TxBufLength = 6 ;

                MisAlignBytes           = (WORD-TxBufLength%WORD)%WORD ;
                psTxBD->BufferDataPtr           = (U32)TxBDataStartAddr & BOwnerShip_CPU;
                psTxBD->Reserved                = 0x0;
                psTxBD->StatusLength  = TxBufLength  ;

                //Next Buffer Descriptor Allocation
                if(psTxBD->PrevBD != (sBufferDescriptor *) NULL){
                        psPrevBD->NextBD = psTxBD ;
                }

                psPrevBD = psTxBD ;
                psTxBD++ ;

                if(i!=MaxTxBDCount) psTxBD->PrevBD = psPrevBD ;
```

```
                TxBDataStartAddr += TxBufLength + MisAlignBytes ;
        }

        psTxBD-- ;
        //Assign last buffer descriptor
        psTxBD->NextBD = psStartBD ;
        //Assign first buffer descriptor
        psStartBD->PrevBD = psTxBD ;
}
```

Listing 3-53. RxBD_init(U8 channel) function (HDLC100INIT.C)

```
/*
 * void RxBD_init(U8 channel) ;
 *       Initialize Rx buffer descriptor.
 */
void RxBD_init(U8 channel)
{
        sBufferDescriptor       *psRxBD ;
        sBufferDescriptor       *psPrevBD, *psStartBD ;
        U32                             RxBDataStartAddr ;
        U32                             i ;

        //RxBBaseAddrA = (U32 *)0x200000 ;

        gCRxBDStart[channel]  = (U32)sRxBDStart[channel] | NonCache ;
        HDMARXPTR(channel)          = gCRxBDStart[channel] ;
        psRxBD                      = (sBufferDescriptor *)gCRxBDStart[channel] ;
        psStartBD                   = psRxBD ;
        psRxBD->PrevBD              = (sBufferDescriptor *)NULL ;
        RxBDataStartAddr            = (U32)RxBBaseAddr[channel] | NonCache ;

        for(i=1; i<=MaxRxBDCount; i++){
                psRxBD->BufferDataPtr       =(U32)RxBDataStartAddr | BOwnerShip_DMA;
                psRxBD->Reserved            = 0x0 ;
                psRxBD->StatusLength = 0x0 ;
                //Next Buffer Descriptor Allocation
                if(psRxBD->PrevBD != (sBufferDescriptor *) NULL)
                        psPrevBD->NextBD = psRxBD ;

                psPrevBD = psRxBD ; // = psRxBD++
                psRxBD++ ;

                if(i!=MaxRxBDCount) psRxBD->PrevBD = psPrevBD ;

                RxBDataStartAddr += RxBufLength ;
        }
        //
        psRxBD-- ;
        //Assign last buffer descriptor
```

```
        psPrevBD->NextBD      = psStartBD ;
        //Assign first buffer descriptor
        psStartBD->PrevBD     = psRxBD ;
}
```

In our diagnostic code , we prepare whole Tx buffer data and controls before operating. TxDataWrite() program do this as described at **Listing 3-54.**.

Listing 3-54. TxDataWrite(U8 channel) function (HDLC100INIT.C)

```
/*
 * void TxDataWrite(U8 channel) ;
 *        Preparing Tx Data to send.
 */
void TxDataWrite(U8 channel,int Owner)
{
        sBufferDescriptor       *psTxBD ;
        U32                             *BufPtr ;
        U32                             *pBDPtr ;
        U32                             i, j, LOOP, TxLength, MisAlignBytes;

        BufPtr = (U32 *) ( (U32) TxBBaseAddr | NonCache ) ;

        psTxBD = sTxBDStart[channel] ;
        psTxBD = (sBufferDescriptor *) ( (U32) psTxBD | NonCache ) ;

        for(i=0; i<MaxTxBDCount; i++){
                pBDPtr = (U32 *)&psTxBD->BufferDataPtr ;
                if(Owner==DMA) *pBDPtr |= BOwnerShip_DMA ;

                psTxBD->Reserved = LastBF;
                TxLength = (U32)psTxBD->StatusLength & 0xFFFFFF ;
                MisAlignBytes = (WORD-TxLength%WORD)%WORD ;
                *BufPtr++ = 0xaaaaaaaa ;
                LOOP = (TxLength<4) ? 0 : TxLength/4-1 ;
                for(j=0; j<LOOP; j++) *BufPtr++ = (j+i) ;
                switch(MisAlignBytes) {
                        case 1 : *BufPtr++ =
                                        (TxLength<<8) & 0x000000cc ; break;
                        case 2 : *BufPtr++ =
                                        (TxLength<<16) & 0x0000cccc ; break;
                        case 3 : *BufPtr++ =
                                        (TxLength<<24) & 0x00cccccc ; break;
                        default : break;
                }
                psTxBD = psTxBD->NextBD ;
        }
}
```

STEP 6 Enable interrupt HDMA Rx and HDLC Rx interrupt

You can refer transmit and receive operation of HDLC in more detail at HDMA Rx, and HDLC Tx interrupt service routine. See **STEP 2.3.**

STEP 7 Interrupt Vector Table Setup and Enabling Interrupt

For normal operation, HDLC diagnostic program uses HDLC Tx/Rx interrupt and HDMA(from here on, HDMA represents the DMA in HDLC). Therefore interrupt mode and interrupt service routine should be initialized. There are four interrupt sources for HDLC which are Tx/Rx interrupts for each A and B channel in KS32C50100. Each HDLC interrupt must be enrolled in interrupt vector table to serve the corresponding interrupt event. The interrupt sources are HDLCTxA_isr(), HDLCRxA_isr(),HDLCTxB_isr() and HDLCRxB_isr(). See **Listing 3-47** and **Listing 3-48.** See **Listing 3-55 , Figure 3-29, Listing 3-56** and **Figure 3-30** for detail s for HDLC interrupt service

Hardware Flow Control

HCON register has two bits to manage Hardware Flow Control. TxDTR bit directly affects the nDTR pin output state if TxEN bit is enabled. When TxDTR bit is cleared, nDTR goes HIGH. And AutoEN bit controls the function of nDCD and nCTS.

The Way to Use Status Register

There are two kinds of bits in status register. One sort of these bits are cleared automatically when this bit indicating status is cleared. These bits are TxFA, TxCTS, RxFA, RxDCD, RxFV, RxCRC, RxNO, RxIERR and RxOV. And the other bits are cleared by writing ‘ 1’ into them using CPU.

If many frames are received, the status bits that indicates Frame valid or Overrun can be set to '1' simultaneously in some case.

The previous frame is valid, but CPU does not handle this valid frame yet, and the current frame is come in Rx FIFO continuously so, Rx FIFO is full, and Rx overrun occurs. The valid frame is moved to memory.

In DMA mode, if Tx underrun bit set to '1', then DMA Tx Abort bit also set to '1'. If Rx Abort bit or RxOverrun is set to '1', then DMA Rx Abort bit also set to '1'. These bits are cleared by CPU writing '1' it. If Tx is enabled, TxFA bit is set to '1' in status register. However, it is not set when Tx is disabled.

When overrun happens, the currently receiving frame is cleared in RxFIFO. Overrun can happen under two circumstances. In the first case of overrun with frame last byte in RxFIFO, the RxOV bit is set to '1' along with RxFV of previously received frame. In the second case of overrun without frame last byte in RxFIFO, the RxOV bit is set to '1' immediately, but the RxFV is not set. The following table show the details on status register. The related bit field means that if status bit field is set , it will be set together normally. And if the affecting bit field is set , the status bit field will meet the setting condition.

Status Bit	Related Bit	Affecting bit	Setting Situation	How to Use	Mode	How to Clear	Causing Interrupt
TxFA	(None)	(None)	When TxFIFO is availabe	When DMA mode , always ‘ 0’	I	After writing data into Tx FIFO	Yes
RxFA	(None)	(None)	When RxFIFO is availabe	When DMA mode , always ‘ 0’	I	After reading data into Rx FIFO	Yes
TxU	TxFC	(None)	run out of data during sending	resending the underrun frame.	D & I	writing ‘ 1’	Yes
TxCTS	TxABT	(None)	When nCTS pin is low	Indicate nCTS pin state	D & I	automatic ally	No
TxSCTS	(None)	(None)	When nCTS transit	Indicate nCTS pin transition occured	D & I	writing ‘ 1’	Yes
DTxABT		TxU	When TxU is set When TxCTS is set to ‘ 0’ during transmission.	DTxEN cleared , So if you want to send another frame , enable DTxEN(HCON[6]) bit.	D	writing ‘ 1’	Yes
		TxCTS					
DTxFD	(None)	(None)	DMA Tx operation is done sucessfully.	Indicate one frame is completed to TxFIFO.	D	writing ‘ 1’	Yes
DTxNL	(None)	(None)	When DMA Tx buffer descriptor pointer has a null list	DTxEN is cleared. So if you want to send another frame, enable DTxEN(HCON[6]) bit.	D	writing ‘ 1’	Yes
DTxNO	(None)	(None)	when DMA is not owner of the current buffer descriptor	If TxSTSK(HCON[14]) was set, DTxEN is cleared. If DRxSTSK was not set, always ‘ 0’ So if you want to send another frame, enable DTxEN(HCON[6]) bit.	D	writing ‘ 1’	Yes
TxFC	(None)	TxU	No data in TxFIFO when meet Closing Flag an abort is transmitted.	Indicate that Frame transmitting is finished. This bit has no information whether the frame is good or not.	D & I	writing ‘ 1’	Yes
RxFD	(None)	(None)	When last bit of the flag sequence is received	Indicate the flag is detected.		writing ‘ 1’	Yes

RxDCD	(None)	(None)	When nDCD pin is low	Indicate nDCD pin state	D & I	automatic ally	No
RxSDCD	(None)	(None)	When nDCD transit	Indicate nDCD pin transition occured	D & I	writing ‘ 1’	Yes
RxFV	(None)	(None)	the last byte of frame is transferred into the last location of the RxFIFO and is available to be read	signals frame’ s ending boundary to CPU and indicate no frame error.	I	After reading data into Rx FIFO	Yes
DPLLOM	(None)	(None)	when doesn’ t detect an edge in two successive attempt with FM/Machester mode	When NRZ/NRZI mode or DPLL disable, always ‘ 0’ . You’ d better clear this bit as soon as possible before two miss occur.		writing ‘ 1’	Yes
DPLLTM	(None)	(None)	when doesn’ t detect an edge in two successive attempt with FM/Machester mode	Enter search mode. When NRZ/NRZI mode or DPLL disable, always ‘ 0’ . HDLC can receive an errored data, so we recommand you discard frame.		writing ‘ 1’	Yes
RxCRCE	(None)	RxIERR	frame is completed with CRC error		D & I	automatic ally	Yes
RxNO	(None)	(None)	received data is non-octet aligned frame		D & I	automatic ally	Yes
RxOV	(None)	(None)	the received data is transferred into the RxFIFO when it is full.	A loss of data . continued overruns destroy data in the first FIFO register	D & I	automatic ally	Yes
RxMOV	(None)	(None)	no more buffer during receiving data	DRxEN is cleared.	D & I	writing ‘ 1’	Yes
DRxFD	(None)	(None)	DMA Rx operation is done successfully.		D	writing ‘ 1’	Yes
DRxNL			When DMA Rx buffer descriptor pointer has a null list	DRxEN is cleared and RxFIFO is cleared. So if you want to receive another frame , enable DRxEN(HCON[7]) bit.	D	writing ‘ 1’	Yes
DRxNO			when DMA is not owner of the	If DRxSTSK (HCON[15]) was set,	D	writing ‘ 1’	Yes

			current buffer descriptor	DRxEN is cleared. If DRxSTSK was not set, always ‘ 0’ . So if you want to receive another frame, enable DRxEN (HCON[7]) bit.			
RxIDLE	(None)	(None)	15 more consecutive 1’ s	Inactive IDLE until ‘ 0’ is received	D & I	writing ‘ 1’	Yes
RxABT	(None)	(None)	7 more consecutive 1’ s		D & I	writing ‘ 1’	Yes
RxIERR	RxNO	(None)	unstable clock	discard the received frame	D & I	automatic ally	Yes
	RxCRCE						

NOTE: D: DMA mode
I: Interrupt mode

Transmit HDLC frame

After setting all control register and HDMA transmit buffer descriptor, you can transmit packet. If you want to 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 HDLC frame data and transmit control function.

STEP 2 Change ownership to HDMA

Because the diagnostic packet of HDLC is ready in buffer data area. You can send this packet by changing ownership to HDMA .

NOTE: If you want to send a newly generated packet , you should set the control bits , buffer data pointer and Lengh at Buffer Descriptor which you want to send.

STEP 3 Enable HDLC and HDMA transmit control register to start transmit.

STEP 4 Change current frame descriptor to next frame descriptor

Listing 3-56 shows that sending transmit packet from start buffer descriptor at the amount of lcount

After HDMA finishing a transmission of a packet , HDLC controller issue HDLC Tx interrupt, in this diagnostic code use two HDLC Tx interrupt for each channel. The HDLC Tx interrupt service routine is only used for check transmit status. The source code for HDLC Tx interrupt service routine is described in **Listing 3-55**.

Listing 3-55. HDLC_Tx_isr( ) function (HDLC100INIT.C)

```
/*
 * Function : HDLC_Tx_isr
 * Description : Interrupt Service Routine for HDLC Tx
 */
void HDLCTxA_isr(void)
{
        U32 IntHDLCStatus ;
        U32 CheckStatus , PacketSize;

        IntHDLCStatus = HSTATA ;
        Clear_PendingBit(nHDLCTxA_INT) ;

        if(IntHDLCStatus & DTxFD)
        {
                gHTxStatus[HDLCA].DMATxFD++ ;
                HSTATA = IntHDLCStatus & DTxFD ;
        }

        CheckStatus = TxSCTS|TxU|DTxABT|DTxNL|DTxNO ;
        if(IntHDLCStatus & CheckStatus){
                if(IntHDLCStatus & (DTxNO|DTxABT|DTxNL)){
                        if(IntHDLCStatus & DTxABT){
                                gHTxStatus[HDLCA].DMATxABT++ ;
                                HSTATA = IntHDLCStatus & DTxABT ;
                        }
                        if(IntHDLCStatus & DTxNO){
                                gHTxStatus[HDLCA].DMATxNO++ ;
                                HSTATA = IntHDLCStatus & DTxNO ;
                        }

                        if(IntHDLCStatus & DTxNL){
                                gHTxStatus[HDLCA].DMATxNL++ ;
                                HSTATA = IntHDLCStatus & DTxNL ;
                        }
                }
        }
        if(!CheckTxStatus(HDLCA))               Print("\r HDLCA Tx Fail ") ;
}
```

```
U32 CheckTxStatus(U8 channel)
{
        sBufferDescriptor       *psTxBDPtr ;
        U32                             *pBDPtr ;
        U32                             CheckHdma ;
        int                             i = 0 ;

        //Check Tx Ownership in all Tx BD
        psTxBDPtr = (sBufferDescriptor *) gCTxBDStart[channel] ;

        while(i < MaxTxBDCount)

                pBDPtr          = (U32 *)&psTxBDPtr->BufferDataPtr ;
                CheckHdma       = (*pBDPtr & BOwnerShip_DMA) ;
                pBDPtr          = (U32 *)&psTxBDPtr->Reserved ;

                if(*pBDPtr & LastBF) {

                        gCTxBDEnd[channel]              = (U32) &psTxBDPtr->BufferDataPtr ;
                        psTxBDPtr                               = (sBufferDescriptor *)gCTxBDEnd[channel] ;
                        gPrevTxBDStart[channel] = gCTxBDStart[channel] ;
                        gCTxBDStart[channel]    = (U32) psTxBDPtr->NextBD ;   //null list

                        pBDPtr                                  = (U32 *) &psTxBDPtr->StatusLength ;
                        //check if transfer is completed
                        CheckHdma                               |= (*pBDPtr & TxComp)^TxComp ;

                        // if last BD Ownership is DMA or Tx Not completed
                        if (CheckHdma) return (U32)HDLCTx_Fail ;
                        else            return (U32)HDLCTx_Ok ;
                }

                if (CheckHdma) // if HDMA Ownership is DMA
                        return (U32)HDLCTx_Fail ;
                else pBDPtr = (U32 *)&psTxBDPtr->NextBD ;
                i++ ;
        }
}
```

The HDLC frame transmit flow is depicted in **Figure 3-29**.

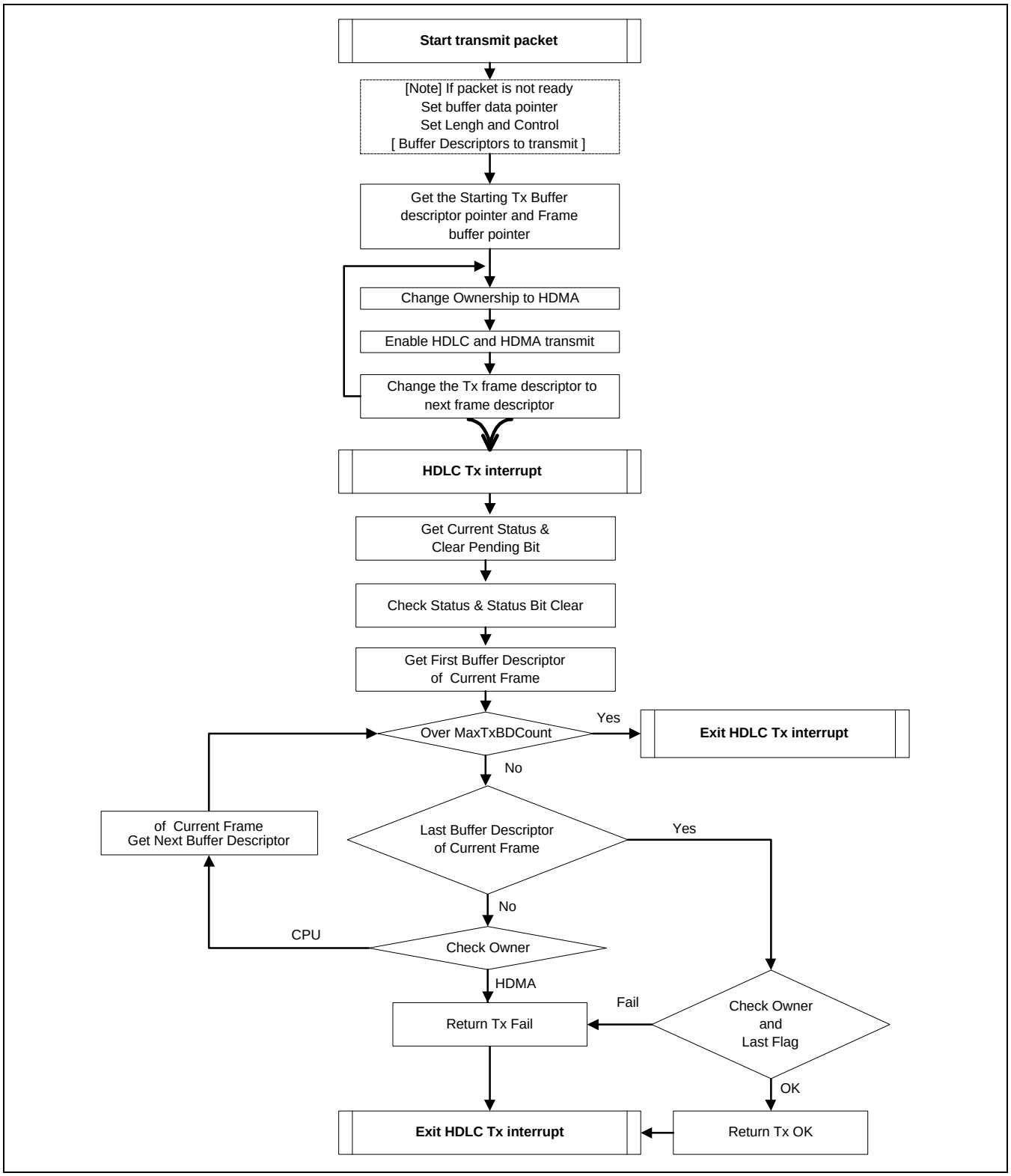


Figure 3-29. HDLC Frame transmit flow

Receive HDLC frame

Receive operation of HDLC frame is performed only on the HDMA Rx interrupt service routine. HDLC Rx interrupt is occurred, when a frame reception is finished. You can follow this step to handling HDLC Rx interrupt.

STEP 1 Clear Pending bit of Interrupt , Get HDLC status and Check HDLC Status

STEP 2 If HDMA Rx Done , Get current descriptor pointer and HDLC status

This step is used for get current frame descriptor pointer from HDMARXPTR register when HDMA Rx Done. The HDMARXPTR register value denote 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 3 Set First buffer descriptor and Last buffer descriptor of the current received frame.

HDMA can consume several buffer descriptors according to the value of HRBSR. So it is needed to check the First and Last flag in buffer descriptor.

STEP 4 Loop from the first buffer descriptor to last buffer descriptor.

This loop is to check one received packet .

STEP 4.1 Check ownership of the current buffer descriptor of the received frame.

In this step we get the receive frame descriptor pointer to process data, every receive process, use HDMA receive frame descriptor pointer.

STEP 4.2 Check status at the current buffer descriptor

If the current buffer has a status error , this frame will not be moved to memory. So return Rx fail.

STEP 4.3 Check First and Last flag set correctly at the current buffer descriptor

If the current buffer has a first and last flag error , this frame will not be moved to memory. So return Rx fail.

STEP 4.4 Get Received buffer data 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 4.4.1 Change ownership to HDMA at current buffer descriptor

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

STEP 4.4.2 Clear status and length field at current buffer descriptor

STEP 5 Check received frame is valid

Confirming the chain , by using FirstFindBD which return the first buffer descriptor of the received frame.
If no error , follow **STEP 6.** If not , return Rx fail
STEP 6 Set first buffer descriptor for a next received frame.

When next Rx interrupt occured , you can find the first buffer descriptor by setting this with address of **STEP 2**

The source code , for the HDLC frame receive operation is described in **Listing 3-7-14**.

Listing 3-56. HDMA_Rx_isr() function (HDLCINIT.C)

```
/*
 * Function : HDMA_Rx_isr
 * Description : Interrupt Service Routine for HDMA Rx
 * Ethenet Frame is received in HDMA_Rx_isr
 */
void HDLCRxA_isr(void)
{
        U32 IntHDLCStatus, pRxBDPtr ;
        U32 RemainByte ;
        IntHDLCStatus = HSTATA ;
        Clear_PendingBit(nHDLCRxA_INT) ;

        if(IntHDLCStatus & DRxFD){
                gHRxStatus[HDLCA].DMARxFD++ ;
                HSTATA = IntHDLCStatus & DRxFD ;

                if(IntHDLCStatus & RxCRCE) {
                        gHRxStatus[HDLCA].RxCRCErr++ ;
                        HSTATA = IntHDLCStatus & DRxFD ;
                }
                pRxBDPtr = HDMARXPTRA ;
                gFrameCount[HDLCA]++;

        }

        if(IntHDLCStatus & (RxMOV|DRxNL|DRxNO)){
                if(IntHDLCStatus & RxMOV) {
                        gHRxStatus[HDLCA].DMARxMOV++ ;
                        HSTATA = IntHDLCStatus & RxMOV ;
                        Print("\r HDLCA Rx Fail");
                        return ;
                }
                if(IntHDLCStatus & DRxNL) {
                        gHRxStatus[HDLCA].DMARxNL++ ;
                        HSTATA = IntHDLCStatus & DRxNL ;
                        Print("\r HDLCA Rx Fail");
                        return ;
                }
                if(IntHDLCStatus & DRxNO) {
                        gHRxStatus[HDLCA].DMARxNO++ ;
                        HSTATA = IntHDLCStatus & RxNO ;
                        Print("\r HDLCA Rx Fail");
                        return ;
                }
        }

        if (!CheckRxStatusHDLC(HDLCA, pRxBDPtr)) Print("\r HDLCA Rx Fail") ;
}
```

```
/*
 * Function : CheckRxStatus
 * Description : In Interrupt Mode, Check Rx Status
 */

U32 CheckRxStatusHDLC(int channel, U32 pRxBDPtr)
{
        sBufferDescriptor       *psRxBDptr;
        U32             *pBDPtr , CheckOwnerShip , RxStatus , nRxBDPtr , FrameStart = 0xFFFF0000 ;
        U32             *first, *last; *CRxBDEnd , *FrameStartAddr ;

        nRxBDPtr                        = pRxBDPtr ;
        FrameStartAddr                  = gUserArea[channel] ;

        while (1) {
                // 1. Get Received Rx Last Buffer Descriptor
                psRxBDptr               = (sBufferDescriptor *)nRxBDPtr ;
                pBDPtr                  = (U32 *)&psRxBDptr->PrevBD ;
                psRxBDptr               = (sBufferDescriptor *)*pBDPtr ;
                pBDPtr                  = (U32 *)&psRxBDptr->BufferDataPtr ;
                CRxBDEnd                = pBDPtr ;
                gCRxBDEnd[channel] = (U32) pBDPtr ;

                first                   = (U32 *)gCRxBDStart[channel] ;
                last                    = (U32 *)nRxBDPtr ;
                psRxBDptr               = (sBufferDescriptor *) gCRxBDStart[channel] ;

                while(psRxBDptr != (sBufferDescriptor *) nRxBDPtr) {
                        pBDPtr          = (U32 *) psRxBDptr ;
                        // 3. Check HDMA Receive Ownership
                        CheckOwnerShip = *pBDPtr & BownerShip_DMA ;
                        if (!CheckOwnerShip)//Ownership is CPU
                                RxStatus = (psRxBDptr->StatusLength >> 16) & 0xffff;
                        else return (U32)HDLCRx_Fail ;

                        if(RxStatus & (Cdlost|CRCE|NonOctet|OverRun|ABT|FLV))          return
(U32)HDLCRx_Fail ;

                        // 4. Check First/Last Flag in BDRxStatus
                        if((pBDPtr!=CRxBDEnd)&(pBDPtr!=first)) {
                                if(RxStatus & (FrameL | FrameF))        return (U32)HDLCRx_Fail ;
                        }
                        else if( pBDPtr==first ) {
                                if( !(RxStatus & FrameF) )              return (U32)HDLCRx_Fail ;
                        }
                        else if( pBDPtr==CRxBDEnd ) {
                                if( !(RxStatus & FrameL) )              return (U32)HDLCRx_Fail ;

                        }
                        psRxBDptr = psRxBDptr->NextBD ;  // To get next buffer
```

```
                    GetRxBDData(channel , U32 pBDPtr);
                }
                gHdlcRxDone = (0x1<<channel) ;

                if( gCRxBDStart[channel] == FindFirstBD((U32)CRxBDEnd) ){
                    gCRxBDStart[channel] = pRxBDPtr ;
                    break ;
                }
                else return (U32) HDLCRx_Fail ;

        } return (U32)HDLCRx_Ok ; // End of while loop
}
```

The receive operation in the HDLC Rx interrupt service routine is shown in **Figure 3-30**.

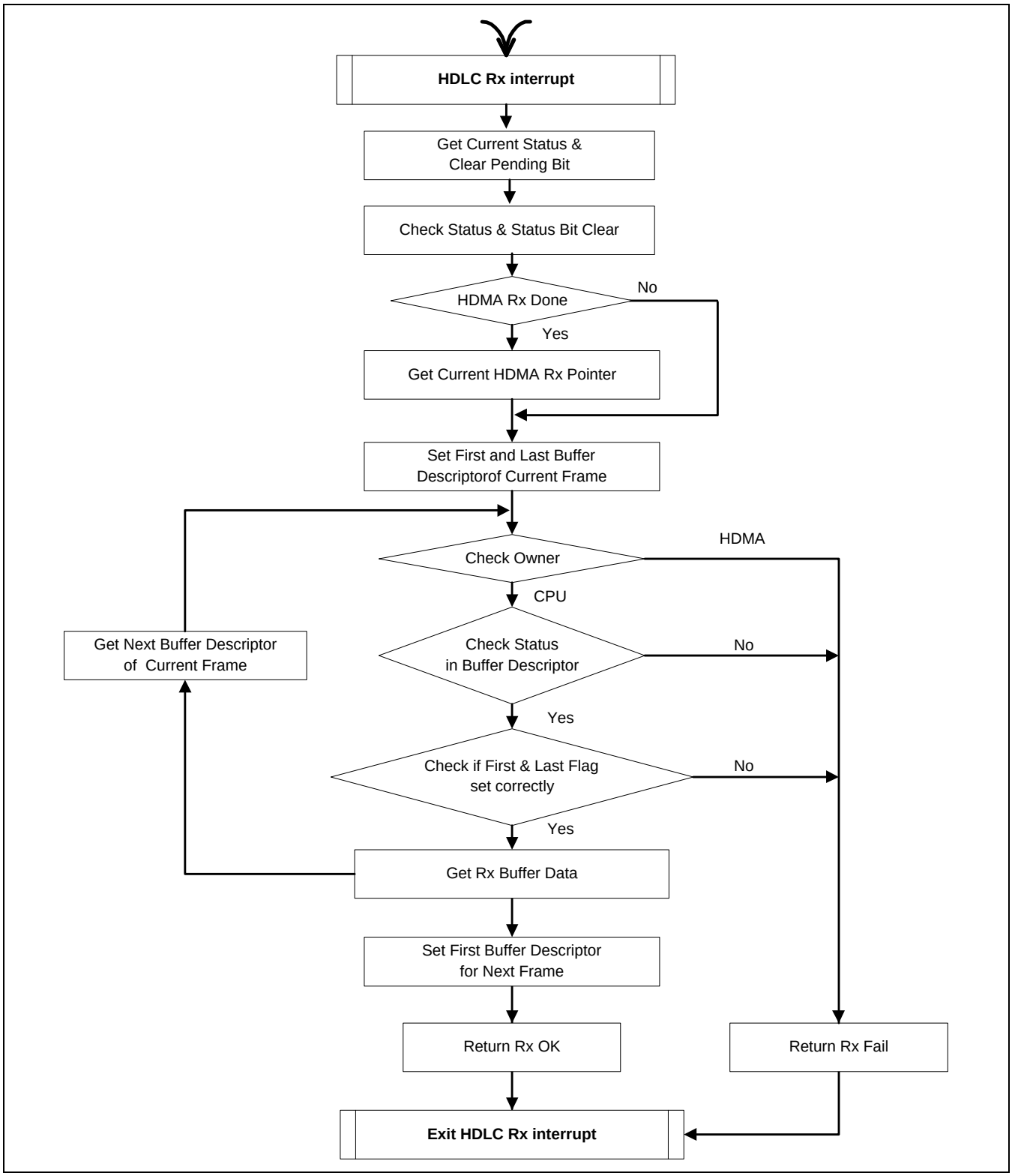


Figure 3-30. HDLC Frame reception flow

Operating on Interrupt mode

If you want to operate HDLC on Interrupt mode , you should initialize registers and interrupt vectors for Interrupt mode which is somewhat different from those of HDMA. Now , we describe only the distinguished point from HDMA mode.

STEP 1 Remove the Bit field related to HDMA mode.

STEP 2 Set HINTEN register as Interrupt mode for your application

→ See the paragraph of **The Way to Use Status Register.**

STEP 3 Program the new Interrupt service routine for Interrupt mode for your application

STEP 4 Program the user routine for TxEnable and Environment setting.

Figure 3-31, and **Listing 3-57** show the operation flow and the sample program.

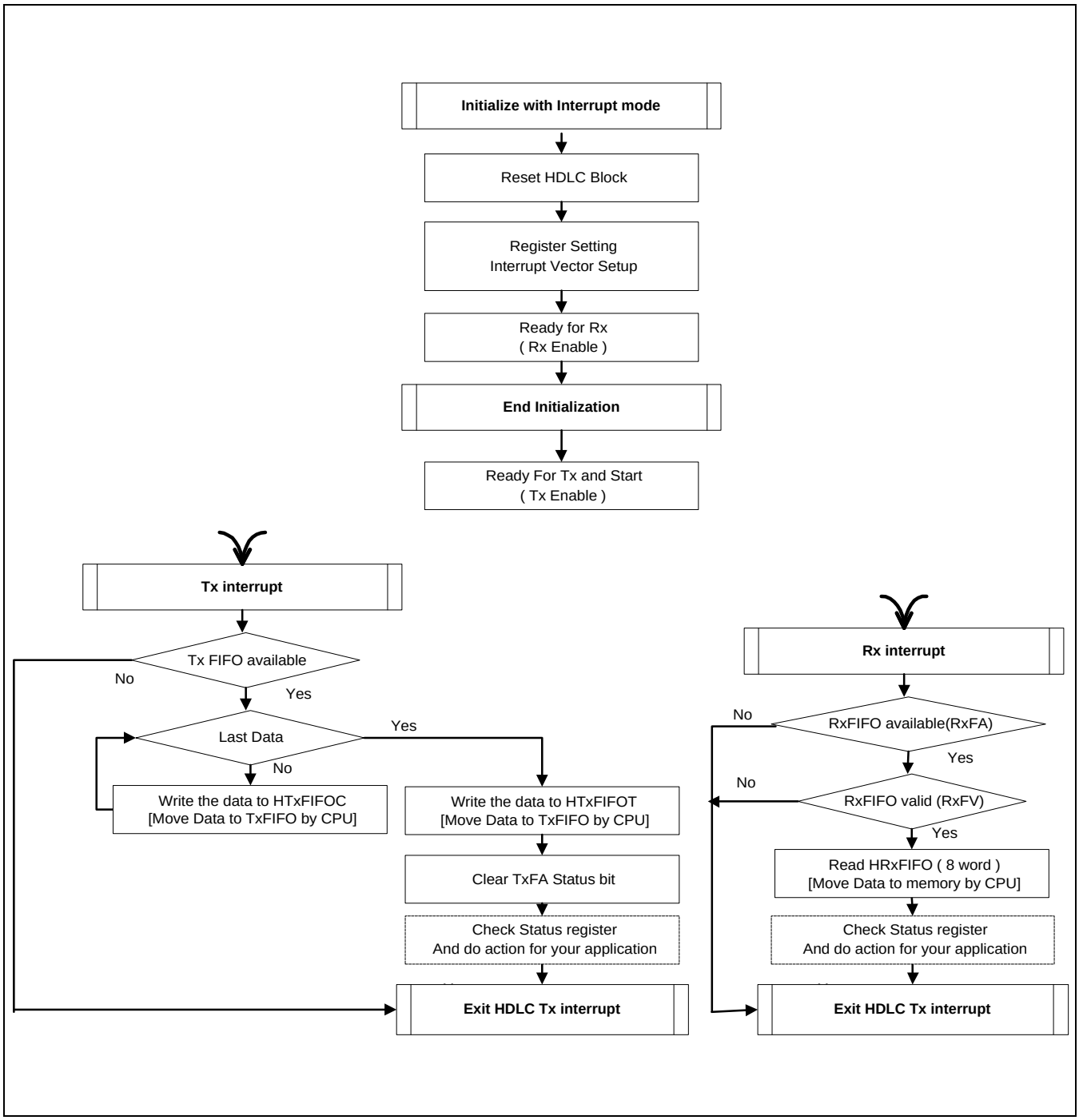


Figure 3-31. Interrupt mode operation flow

Listing 3-57. Interrupt mode functions

```
/*
 * Function      : HdlcInitialize
 * Description   : This function initialize the HDLC Block For Interrupt mode
 */
void HdlcInitialize (void)
{
        U8      channel;        U32     UserArea;

        for(channel=HDLCA;channel<=HDLCB;channel++)
        {
                //Reset All HDLC block
                HdlcReset(channel) ;

                // Internal Register Setting for Tx/Rx operation
                HMODE(channel)          = gMode  = NRZ | TxCBO2 | RxCRxC | BRGMCLK | TxOTxC ;
                HCON(channel)           = gControl = Tx1WD|Rx1WD|TxFlag|TxDTR | BRGEN ;
                HINTEN(channel)   = gHIntEN = TxFCIE | TxFAIE | TxUIE | RxFAIE | RxFVIE | RxABTIE |
RxCRCEIE | RxNOIE | RxOVIE             HBRGTC(channel)  = GetBaudRate(HDLCBaud) ;
                AddressSet(channel) ;
                HDLCRxEN(channel) ;

        }

        // Interrupt Vector Setup
        SysSetInterrupt(nHDLCTxA_INT, HDLCTxA_isr );
        SysSetInterrupt(nHDLCRxA_INT, HDLCRxA_isr );
        SysSetInterrupt(nHDLCTxB_INT, HDLCTxB_isr );
        SysSetInterrupt(nHDLCRxB_INT, HDLCRxB_isr );

        Enable_Int(nHDLCTxA_INT);
        Enable_Int(nHDLCRxA_INT);
        Enable_Int(nHDLCTxB_INT);
        Enable_Int(nHDLCRxB_INT);
        Enable_Int(nGLOBAL_INT);

        // Initialize Global Variables
}

/*
 * Function : HDLC_Tx_isr
 * Description : Interrupt Service Routine for HDLC Tx
 */
void HDLCTxA_isr(void)
{
         if(IntHDLCStatus & TxFA){
                for(i=0; i<gMaxPacketCnt; i++){
                   PacketSize = PatternGen(i)&0xFFFF;
                        if(PacketSize < 2) PacketSize = 2 ;

                        HTXFIFOCA = 0xFFFF0000 | (i+1) ;
```

```
                        for(j=0; j<PacketSize; j++){
                                if(j==(PacketSize-1)){
                                        HTXFIFOTA = 0xEEEE0000|PacketSize ;
                                } else HTXFIFOCA = PatternGen(j) | i ;
                        }

                        if(i==(gMaxPacketCnt-1))
                                HINTENA &= ~ TxFAIE ;
                }
        }

        if(IntHDLCStatus & TxFC){
                gHTxStatus[HDLCA].TxFrameComp++ ;
        }
        if(IntHDLCStatus & TxU){
                gHTxStatus[HDLCA].TxUnderrun++ ;
        }
}

/*
 * Function : HDLC_Rx_isr
 * Description : Interrupt Service Routine for HDLC Rx
 */
void HDLCRxA_isr(void)
{
        if(IntHDLCStatus & (RxFA|RxFV))
        {
                if(IntHDLCStatus & RxFA)
                  *gUserArea[HDLCA]++ = HRXFIFOB ;
                if(IntHDLCStatus & RxFV)
                {
                        gHRxStatus[HDLCA].RxFValid++ ;
                        RemainByte = IntHDLCStatus & 0xf;

                        if(RemainByte ==  RxRB1)
                                *gUserArea[HDLCA]++ = HRXFIFOB & 0xFF000000 ;
                        if(RemainByte ==  RxRB2)
                                *gUserArea[HDLCA]++ = HRXFIFOB & 0xFFFF0000 ;
                        if(RemainByte ==  RxRB3)
                                *gUserArea[HDLCA]++ = HRXFIFOB & 0xFFFFFF00 ;
                        if(RemainByte ==  RxRB4)
                                *gUserArea[HDLCA]++ = HRXFIFOB & 0xFFFFFFFF ;
                        if(!CheckRxCPU()) {
                                RxFail = 1 ;
                        }
                        PrintMemTestStatus(RemainByte%4) ;
                }
                //if(IntHDLCStatus & RxFA)
                  //*gUserAreaB++ = HRXFIFOB ;


        if(IntHDLCStatus & RxCRCE) {
```

```
                gHRxStatus[HDLCB].RxCRCErr++ ;
                *gUserArea[HDLCB]++ = 0x0000ffff & HRXFIFOB ;
                RxFail = 1 ;
        }
        if(IntHDLCStatus & RxOV) {
                gHRxStatus[HDLCB].RxOverrun++ ;
                put_byte('O') ;
                RxFail = 1 ;
        }
        if (RxFail) Print("\r HDLCB Rx Fail") ;
   }
}

void CPUTxAGo (void)
{
        U32 i ;

        // Setting for diagnostic test
        gMaxPacketCnt = 20 ;
        gUserArea[HDLCB] = (U32 *)0x1250000 ;
        gCPUmode = 1 ;

        // Clear User Area
        for(i=0; i<100000; i++)
                *gUserArea[HDLCB]++ = 0x0 ;

        HMASKA = 0x00000000 ;
        HDLCLoopbackEn(HDLCB) ;

        Print("\nHDLC CPU mode CH B internal LoopBack Test .... ") ;

        // Tx Enable
        HDLCTxEN(HDLCB) ;
}
```

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, get MAC H/W address from IIC EEPROM, 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, 10/100Mbps, and Full/Half duplex mode.

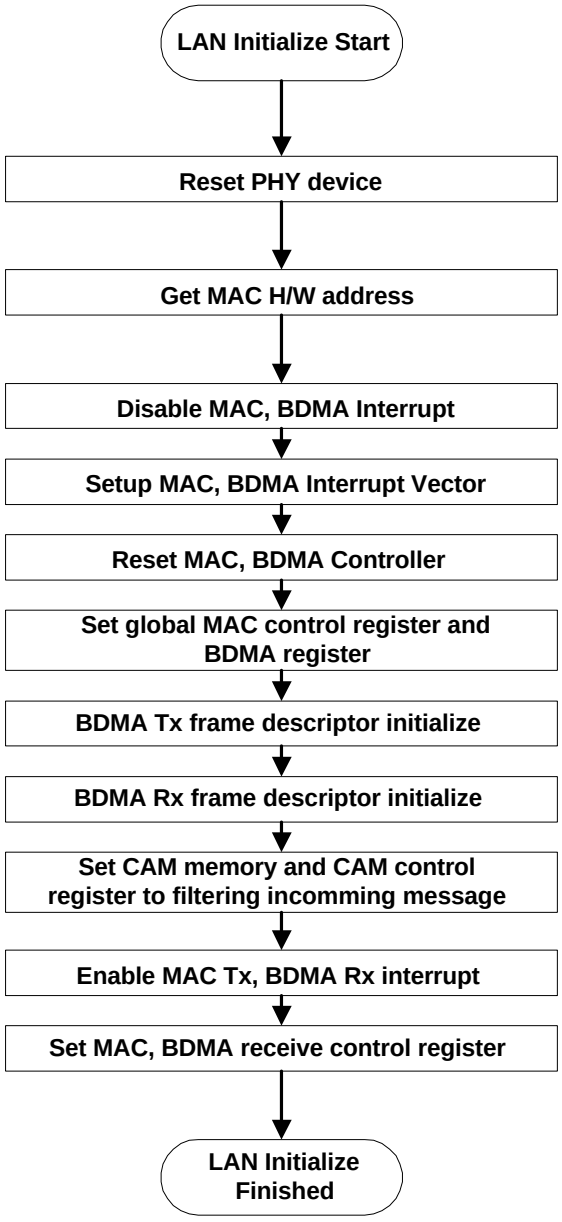
The MAC controller has MAC H/W address, this address is unique address for each MAC, so you need to set the MAC H/W address for each MAC. In SNDS and SNDS100 board use IIC EEPROM to store MAC H/W address.

The KS32C5000(A)/KS32C50100 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.

The detail of LAN initialize function is described in source code **listing 3-58** and **figure 3-32**.

Listing 3-58. LanInitialize() function (MACINIT.C)

```
/*
 * void LanInitialize(void)
 */
void LanInitialize(void)
{
        // Reset the PHY chip and start Auto-negotiat
        ResetPhyChip() ;
        // Read MAC address from IIC EEPROM
        GetMyMacAddr() ;
        // Initialize MAC and BDMA controller
        MacInitialize() ;
}
```



1. Reset PHY device and configure via MII station Management control function

2. Get MAC H/W unique address
   Stored on IIC EEPROM

3. Setup MAC, BDMA controller
   1) Disable MAC, BDMA Tx/Rx interrupt
   2) Setup MAC, BDMA interrupt vector
   3) Reset MAC, BDMA controller
   4) Set global MAC control register and BDMALSZ register
   5) BDMA Tx Frame descriptor initialize
   6) BDMA Rx Frame descriptor initialize
   7) Set CAM memory and control register to filtering incomming message
   8) Enable MAC Tx, BDMA Rx interrupt
      We use MAC Tx interrupt for transmit operation, and BDMA Rx interrupt for receive operation.
   9) Set MAC, BDMA receive control register

Figure 3-32. 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 address for PHY internal control and status register.

The operation of read, and write to the station management register is described in below. And the source code for station management register read and write function is described in **Listing 3-59**.

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.

Listing 3-59. MiiStationRead(), MiiStationWrite() function (MACINIT.C)

```
/*
 * Function : MiiStationRead, MiiStationWrite
 * Description : MII Interface Station Management Register Read or Write
 * Input : PhyInAddr(PHY internal register address)
 *         PhyAddr(PHY unique address)
 *         PhyWrData(When Write)
 * Output: PhyRdData(WhenRead)
 */
void MiiStationWrite(U32 PhyInAddr, U32 PhyAddr, U32 PhyWrData)
{
        STADATA = PhyWrData ;
        STACON = PhyInAddr | PhyAddr |  MiiBusy | PHYREGWRITE ;
        while( (STACON & MiiBusy) )  ;
        delay_physet() ;
}
U32 MiiStationRead(U32 PhyInAddr, U32 PhyAddr)
{
        U32     PhyRdData ;
        STACON = PhyInAddr | PhyAddr |  MiiBusy ;
        while( (STACON & MiiBusy) )  ;
        PhyRdData = STADATA ;
        return PhyRdData ;
}
```

The sample code for configuring the PHY device is

Listing 3-60. ResetPhyChip() function (MACINIT.C)

```
/*
 * Function : ResetPhyChip
 * Description : Reset The Phychip, Auto-Negotiation Enable
 */
void ResetPhyChip(void)
{
        //MiiStationWrite(PHY_CNTL_REG, PHYHWADDR, ENABLE_AN | RESTART_AN);  // Auto-negotion
enable
        //MiiStationWrite(PHY_CNTL_REG,PHYHWADDR, 0x0) ;                    // 10Mbps, Half-duplex mode
        //MiiStationWrite(PHY_CNTL_REG,PHYHWADDR,PHY_FULLDUPLEX) ;         // 10Mbps, Full-duplex
mode
        //MiiStationWrite(PHY_CNTL_REG,PHYHWADDR,DR_100MB) ;               // 100Mbps, Half-duplex
        MiiStationWrite(PHY_CNTL_REG,PHYHWADDR,DR_100MB|PHY_FULLDUPLEX) ; // 100Mbps, Full-
duplex mode
        gDuplexValue = FullDup ; // This variable shouldbe set when use Full-duplex mode.
}
```

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 SNDS and SNDS100 use IIC EEPROM for store MAC H/W address for system.

Listing 3-61. GetMyMacAddr() function (MACINIT.C)

```
 /*
 * Function : GetMyMacAddr
 * Description : Get MAC Address From IIC EEPROM
 */
void GetMyMacAddr(void)
{
        U8 *MacAddr;
        U8 TempAddr[MAC_ADDR_SIZE];
        int i;

        gCam0_Addr0 = gCam0_Addr1 = 0 ;

#if MAC_ADDR_FROM_IIC_EEPROM // when use IIC EEPROM for store MAC address
        MacAddr=(U8 *)IICReadInt((U8)IIC_DEV_0,(U32)iMacAddrPtr,(U32)MAC_ADDR_SIZE) ;
#else   // When manual ly setting the MAC address
        MacAddr[0] = 0x00 ;
        MacAddr[1] = 0x00 ;
        MacAddr[2] = 0xf0 ;
        MacAddr[3] = 0x11 ;
        MacAddr[4] = 0x00 ;
        MacAddr[5] = 0x00 ;
#endif

        /* Get MAC Address */
```

```
        for (i=0;i<(int)MAC_ADDR_SIZE;i++)  TempAddr[i] = *MacAddr++;

        /* Copy MAC Address to global variable */
        for (i=0;i<(int)MAC_ADDR_SIZE-2;i++)
                gCam0_Addr0 = (gCam0_Addr0 << 8) | TempAddr[i] ;

        for (i=(int)(MAC_ADDR_SIZE-2);i<(int)MAC_ADDR_SIZE;i++)
                gCam0_Addr1 = (gCam0_Addr1 << 8) | TempAddr[i] ;
        gCam0_Addr1 = (gCam0_Addr1 << 16) ;

        /* Set CAM0 register : 0x9100~0x9103, 0x9104, 0x9105 */
    CAM_Reg(0) =  gCam0_Addr0;
    CAM_Reg(1) =  gCam0_Addr1;

        for (i=0;i<(int)MAC_ADDR_SIZE;i++) MyMacSrcAddr[i] = TempAddr[i];
}
```

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 main function for setup MAC, and BDMA controller is described in **Listing 3-62**.

Listing 3-62. MacInitialize() function (MACINIT.C)

```
/*
 * void MacInitialize(void)
 * Initialize MAC and BDMA Controller
 */
void MacInitialize(void)
{

        /*--------------------------------------------------------------*
         * 1. Disable MAC and BDMA interrupts.          *
         *--------------------------------------------------------------*/
        Disable_Int(nMAC_RX_INT) ;     // Disable MAC Rx Interrupt
        Disable_Int(nMAC_TX_INT) ;     // Disable MAC Tx Interrupt
        Disable_Int(nBDMA_RX_INT) ;    // Disable BDMA Rx Interrupt
        Disable_Int(nBDMA_TX_INT) ;    // Disable BDMA Tx Interrupt

        /*--------------------------------------------------------------*
         * 2. BDMA and MAC interrupt vector setup.        *
         *--------------------------------------------------------------*/
        SysSetInterrupt(nMAC_RX_INT, MAC_Rx_isr) ;
        SysSetInterrupt(nMAC_TX_INT, MAC_Tx_isr) ;
        SysSetInterrupt(nBDMA_RX_INT, BDMA_Rx_isr) ;
        SysSetInterrupt(nBDMA_TX_INT, BDMA_Tx_isr) ;


        /*--------------------------------------------------------------*
         * 3. Set the initial condition of the BDMA, and MAC  *
         *--------------------------------------------------------------*/
        BDMARXCON = BRxRS ;            // Reset BDMAC Receiver
```

```
        BDMATXCON = BTxRS ;             // Reset BDMAC Transceiver
        MACCON = SwReset ;
        BDMARXLSZ = MaxRxFrameSize ;    // 1520
        MACCON = gMacCon ;

        /*--------------------------------------------------------------*
         * 4. Set the BDMA Tx/Rx Frame Descriptor        *
         *--------------------------------------------------------------*/
        TxFDInitialize() ;
        RxFDInitialize() ;

        /*--------------------------------------------------------------*
         * 5. Set the CAM Control register and the MAC address value      *
         *--------------------------------------------------------------*/
        // CAM0 register of CAM registers : 0x9100~0x9103, 0x9104, 0x9105
        CAM_Reg(0) = gCam0_Addr0 ;
        CAM_Reg(1) = gCam0_Addr1 ;

        // CAM Enable Register(CAMEN)
        CAMEN = 0x0001 ;
        CAMCON = gCamCon ;

        /*--------------------------------------------------------------*
         * 6. Enable interrupt BDMA Rx and MAC Tx interrupt. *
         *--------------------------------------------------------------*/
        Enable_Int(nBDMA_RX_INT);
        Enable_Int(nMAC_TX_INT);

        /*--------------------------------------------------------------*
         * 7. Configure the BDMA and MAC control registers.  *
         *--------------------------------------------------------------*/
        ReadyMacTx() ;
        ReadyMacRx() ;
}
```

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

STEP 1 Disable MAC and BDMA interrupt

Disable MAC, BDMA Tx/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.

The frame descriptor structure is described in **Listing 3-63**, and **Figure 3-33**.

Listing 3-63. Frame Descriptor Structure (MAC.H)

```
// Tx/Rx common descriptor structure
typedef struct FrameDescriptor {
        U32 FrameDataPtr;
        U32 Reserved;  // cf: RX-reserved, TX-Reserved(25bits) + Control bits(7bits)
        U32 StatusAndFrameLength;
        U32 NextFrameDescriptor;
} sFrameDescriptor;
```

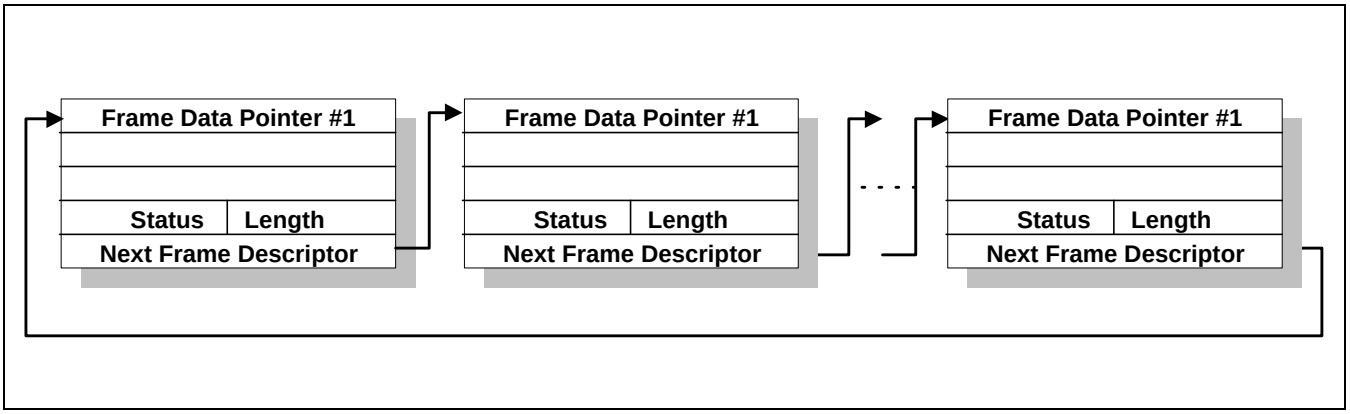


Figure 3-33. BDMA Frame Descriptor Structure

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 **Listing 3-64**, and **Listing 3-65** show the detail operation of initialize BDMA Tx/Rx frame descriptor.

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.

Listing 3-64. TxFDInitialize() function (MACINIT.C)

```
/*
 * void TxFDInitialize(void) ;
 *          Initialize Tx frame descriptor area-buffers.
 */
void TxFDInitialize(void)
{
        sFrameDescriptor *pFrameDescriptor;
        sFrameDescriptor *pStartFrameDescriptor;
        sFrameDescriptor *pLastFrameDescriptor = NULL;
        U32 FrameDataAddr;
        U32 i;

        // Get Frame descriptor's base address.
        // +0x4000000 is for setting this area to non-cacheable area.
        BDMATXPTR = (U32)TxFDBaseAddr + 0x4000000;
        gWTxFDPtr = gCTxFDPtr = BDMATXPTR;

        // Get Transmit buffer base address.
        FrameDataAddr = (U32)TxFBABaseAddr + 0x4000000;

        // Generate linked list.
        pFrameDescriptor = (sFrameDescriptor *) gCTxFDPtr;
        pStartFrameDescriptor = pFrameDescriptor;

        for(i=0; i < MaxTxFrameDescriptors; I++)  {
                if(pLastFrameDescriptor == NULL)
                        pLastFrameDescriptor = pFrameDescriptor;
                else    pLastFrameDescriptor->NextFrameDescriptor = (U32)pFrameDescriptor;

                pFrameDescriptor->FrameDataPtr =
                        (U32)(FrameDataAddr & fOwnership_CPU);
                pFrameDescriptor->Reserved = (U32)0x0;
                pFrameDescriptor->StatusAndFrameLength = (U32)0x0;
                pFrameDescriptor->NextFrameDescriptor = NULL;

                pLastFrameDescriptor = pFrameDescriptor;
                pFrameDescriptor++;
                FrameDataAddr += sizeof(sMACFrame);
        } // end for loop

        // Make Frame descriptor to ring buffer type.
        pFrameDescriptor--;
        pFrameDescriptor->NextFrameDescriptor = (U32)pStartFrameDescriptor;
}
```

Listing 3-65. RxFDInitialize() function (MACINIT.C)

```
/*
 * void RxFDInitialize(void) ;
 *          Initialize Rx frame descriptor area-buffers.
 */
void RxFDInitialize(void)
{
        sFrameDescriptor *pFrameDescriptor;
        sFrameDescriptor *pStartFrameDescriptor;
        sFrameDescriptor *pLastFrameDescriptor = NULL;
        U32 FrameDataAddr;
        U32 i;

        // Get Frame descriptor's base address.
        // +0x4000000 is for setting this area to non-cacheable area.
        BDMARXPTR = (U32)RxFDBaseAddr + 0x4000000;
        gCRxFDPtr = BDMARXPTR;

        // Get Transmit buffer base address.
        FrameDataAddr = (U32)RxFBABaseAddr + 0x4000000;

        // Generate linked list.
        pFrameDescriptor = (sFrameDescriptor *) gCRxFDPtr;
        pStartFrameDescriptor = pFrameDescriptor;

        for(i=0; i < MaxRxFrameDescriptors; i++)  {
                if(pLastFrameDescriptor == NULL)
                        pLastFrameDescriptor = pFrameDescriptor;
                else    pLastFrameDescriptor->NextFrameDescriptor = (U32)pFrameDescriptor;

                pFrameDescriptor->FrameDataPtr =
                (U32)(FrameDataAddr | fOwnership_BDMA | 0x4000000 );
                pFrameDescriptor->Reserved = (U32)0x0;
                pFrameDescriptor->StatusAndFrameLength = (U32)0x0;
                pFrameDescriptor->NextFrameDescriptor = NULL;

                pLastFrameDescriptor = pFrameDescriptor;
                pFrameDescriptor++;
                FrameDataAddr += sizeof(sMACFrame);
        } // end for loop

        // Make Frame descriptor to ring buffer type.
        pFrameDescriptor--;
        pFrameDescriptor->NextFrameDescriptor = (U32)pStartFrameDescriptor;
}
```

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

CAM is used for filtering received frame from other frames. The KS32C5000(A)/KS32C50100 has 21 CAM, but in the diagnostic code uses only 1 CAM. In diagnostic code for SNDS and SNDS100 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** 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 3-6**.

Table 3-6. 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	EnComp	Interrupt when MAC Tx is completed or discarded
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(16words)
		BTxMSL110	BDMA Tx to MAC Tx start level(6/8)
		BTxSTSKO	BDMA Tx stop when met not owned frame
BDMARXCON	BDMA Receive control register	BRxDIE	Interrupt on every received frame
		BRxEN	BDMA receive enable
		BRxLittle	Received data is stored in Little-endian mode (Used when Little-endian is selected)
		BRxBig	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(16words)
		BRxNLIE	BDMA Rx null list interrupt enable
		BRxNOIE	BDMA Rx not owner interrupt enable
		BRxSTSKO	BDMA Rx stop when met not owned frame

The source code for setup MAC and BDMA control register is shown in **Listing 3-66, Listing 3-67**.

Listing 3-66. Global variable for MAC and BDMA control register (MACINIT.C)

```
// Global variables  used for MAC driver
volatile U32 gMacCon = FullDup ;
volatile U32 gMacTxCon = EnComp ;
volatile U32 gMacRxCon = RxEn | StripCRC ;
volatile U32 gBdmaTxCon = BTxBRST | BTxMSL110 | BTxSTSKO ;
#ifdef LITTLE
volatile U32 gBdmaRxCon = BRxDIE | BrxEn | BRxLittle | BrxMAINC | BRxBRST | \
                BRxNLIE | BRxNOIE | BRxSTSKO ;
#else
volatile U32 gBdmaRxCon = BRxDIE | BrxEn | BRxBig | BRxMAINC | BRxBRST | \
                BRxNLIE | BRxNOIE | BRxSTSKO ;
#endif
volatile U32 gCamCon = CompEn| BroadAcc;
```

Listing 3-67. ReadyMacTx() and ReadyMacRx() function (MACINIT.C)

```
 /*
 * Function : ReadyMacTx
 * Description : set Tx Registers related with BDMA & MAC to transmit
 *               packet.
 */
void ReadyMacTx(void)
{
        BDMATXCON = gBdmaTxCon ;
        MACTXCON =  gMacTxCon ;
}

/*
 * Function : ReadyMacRx
 * Description : set Rx Registers related with BDMA & MAC to Receive packet.
 */
void ReadyMacRx(void)
{
        BDMARXCON = gBdmaRxCon ;
        MACRXCON = gMacRxCon ;
}
```

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 source code for transmit packet is described in **Listing 3-68**.

After transmit Ethernet frame MAC controller issue MAC Tx interrupt, in this diagnostic code only use MAC Tx interrupt for transmit. The MAC Tx interrupt service routine is only used for check transmit status. The source code for MAC Tx interrupt service routine is described in **Listing 3-69**.

Listing 3-68. SendPacket( ) function (MACINIT.C)

```
/*
 * Function : SendPacket
 * Description : Send ethernet frame function
 * Input : frame data pointer, frame length
 * Output : transmit ok(1) or error(0)
 */
int SendPacket(U8 *Data,int Size)
{
        sFrameDescriptor        *psTxFD;
        U32                     *pFrameDataPtr ;
        U8                      *pFrameData ;
        int                     FrameLength ;

        // 1. Get Tx frame descriptor & data pointer
```

```
        psTxFD = (sFrameDescriptor *)gWTxFDPtr ;

        pFrameData = (U8 *)psTxFD->FrameDataPtr ;
        pFrameDataPtr = (U32 *)&psTxFD->FrameDataPtr;
        FrameLength = Size + sizeof(etheader) ;

        // 2. Check BDMA ownership
        if ( (*pFrameDataPtr & fOwnership_BDMA) ) return 0 ;

        // 3. Prepare Tx Frame data to Frame buffer
        memcpy ((U8 *)pFrameData,(U8 *)Data,FrameLength);

        // 4. Set TX Frame flag & Length Field
#ifdef LITTLE
        psTxFD->Reserved = (PaddingMode | CRCMode | SourceAddrIncrement | \
                        LittleEndian | WidgetAlign00 | MACTxIntEn);
#else
        psTxFD->Reserved = (PaddingMode | CRCMode | SourceAddrIncrement | \
                        BigEndian | WidgetAlign00 | MACTxIntEn);
#endif
        psTxFD->StatusAndFrameLength = (U32)(FrameLength & 0xffff);

        // 5. Cheange ownership to BDMA
        psTxFD->FrameDataPtr |= fOwnership_BDMA;

        // 6. Enable MAC and BDMA Tx control register
        MacTxGo();

        // 7. Change the Tx frame descriptor for next use
        gWTxFDPtr = (U32)(psTxFD->NextFrameDescriptor);
        return 1 ;
}
```

Listing 3-69. MAC_Tx_isr() function (MACINIT.C)

```
/*
 * Function : MAC_Tx_isr
 * Description : Interrupt Service Routine for MAC Tx
 */
void MAC_Tx_isr(void)
{
        sFrameDescriptor *pTxFDptr;
        U32 *pFrameDataPtr ;
        U32 Status ;
        U32 CTxPtr ;

        CTxPtr = BDMATXPTR ;
        while ( gCTxFDPtr != CTxPtr )
                pTxFDptr = (sFrameDescriptor *) gCTxFDPtr;
```

```
                // Check CPU ownership
                // if Owner is BDMA then break
                pFrameDataPtr = (U32 *)&pTxFDptr->FrameDataPtr;
                if ( (*pFrameDataPtr & fOwnership_BDMA) ) break ;

                Status = (pTxFDptr->StatusAndFrameLength >> 16) & 0xffff;

                if (Status & Comp) gsMacTxStatus.MacTxGood++ ;
                else
                {
                        // Save Error status
                        // Check each status, because, error can duplicated
                        if (Status & Under) gsMacTxStatus.UnderErr++ ;
                        if (Status & ExColl) gsMacTxStatus.ExCollErr++ ;
                        if (Status & TxDeffer) gsMacTxStatus.TxDefferedErr++ ;
                        if (Status & Paused) gsMacTxStatus.sPaused++ ;
                        if (Status & Defer) gsMacTxStatus.DeferErr++ ;
                        if (Status & NCarr) gsMacTxStatus.NCarrErr++ ;
                        if (Status & SQErr) gsMacTxStatus.sSQE++ ;
                        if (Status & LateColl) gsMacTxStatus.LateCollErr++ ;
                        if (Status & TxPar) gsMacTxStatus.TxParErr++ ;
                        if (Status & TxHalted) gsMacTxStatus.sTxHalted++ ;

                        // Set MAC/BDMA Tx control register for next use.
                        MacDebugStatus() ;
                        ReadyMacTx() ;
                } // end if

                // Clear Framedata pointer already used.
                pTxFDptr->StatusAndFrameLength = (U32)0x0;

                gCTxFDPtr = (U32)pTxFDptr->NextFrameDescriptor ;
        } // end while loop

        MacTxDoneFlagForLoopBackCheck = 1 ;
}
```

The Ethernet frame transmit flow is depicted in **Figure 3-34**.

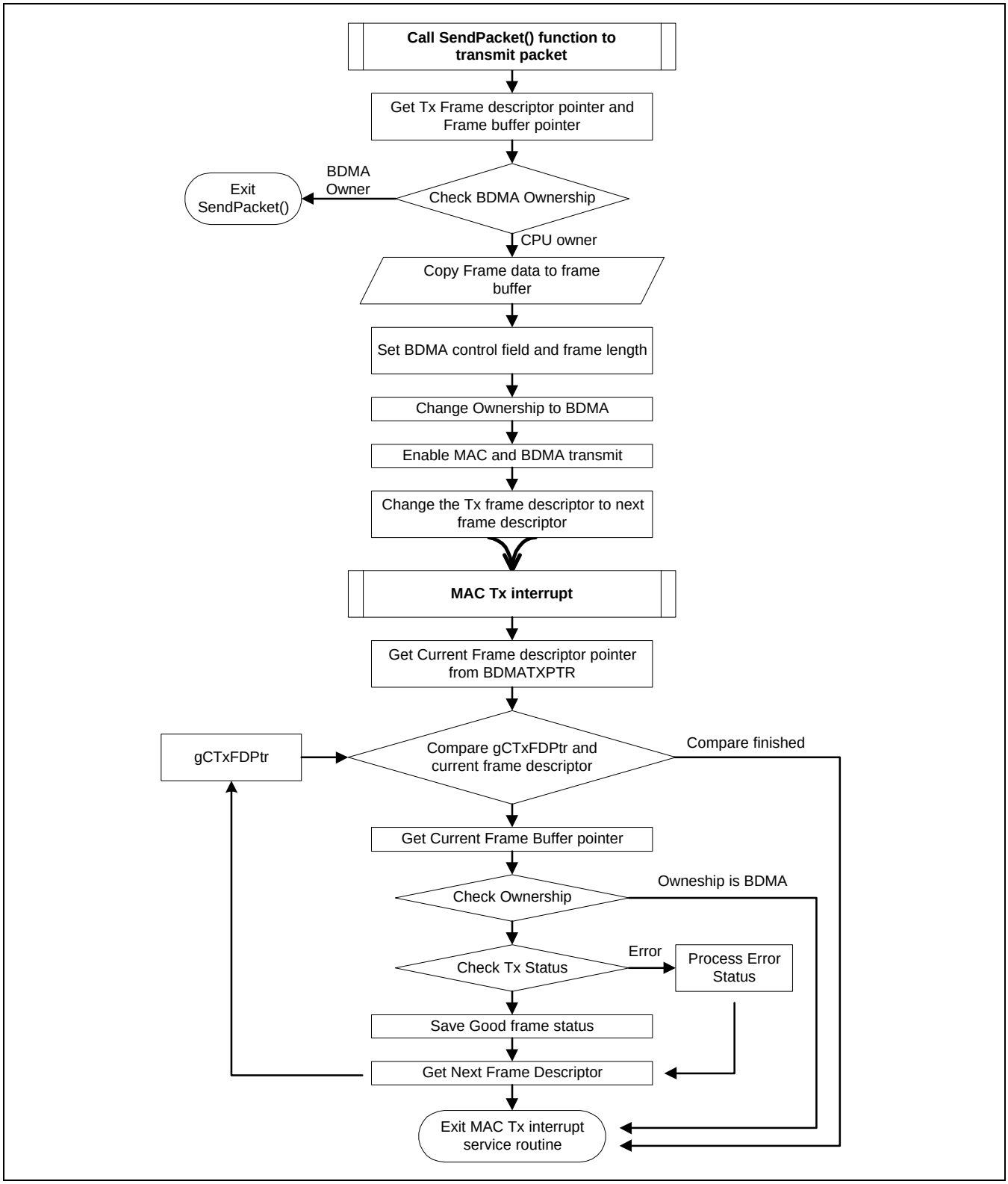


Figure 3-34. Ethernet Frame transmit flow

Control Frame Transmit

You can transmit a control frame for a remote pause operation in full-duplex mode operation. KS32C5000(A)/KS32C50100 Ethernet controller has a function of transmitting and receiving control frame. When transmit control frame, follow this step. The source code is described in **Listing 3-70**.

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.

Listing 3-70. ControlFrameTransfer() function (MACINIT.C)

```
/*
 * Function : ControlFrameTransfer
 * Description : Transfer Control Frame Data to another Host
 */
void ControlFrameTransfer(void)
{
        char TransferPacket ;

        gBdmaTxCon  |= BTxCPIE ;
        ReadyMacTx() ;

        do
        {
        Print("\r   $$ Select Transmit(T) or Quit(Q) ? ") ;
        TransferPacket = get_byte() ;
        if ( (TransferPacket == 'T') || (TransferPacket == 't') )
                {

                // Step 1. set destination address to CAM #0
                // Step 2. set source address to CAM #1
                // CAM #0 : 0000f0110000 (Destin Addr)
                // CAM #1 : 11f000000000 (Source Addr)
                VPint(CAM_BaseAddr) = 0x0000f011 ;
                VPint(CAM_BaseAddr + 0x4) = 0x000011f0 ;
                VPint(CAM_BaseAddr + 0x8) = 0x00000000 ;

                // Step 3. set length or type field, opcode, and operand to CAM #18
                // CAM #18 : Opcode & Operand
                // CAM #18 : Pause Count Value
                VPint(CAM_BaseAddr + 0x6c) = 0x88080001 ;

                // Step 4. set to zero proceed CAM #18
                // CAM #19 - #20 : Filled with Zero
                VPint(CAM_BaseAddr + 0x70) = 0x12340000 ;
                VPint(CAM_BaseAddr + 0x74) = 0x00000000 ;
                VPint(CAM_BaseAddr + 0x78) = 0x00000000 ;
```

```
                VPint(CAM_BaseAddr + 0x7c) = 0x00000000 ;

                // Step 5. Enable CAM location
                // CAM Enable
                CAMEN = 0x1c0003 ;

                // Step 6. Enable transmit control frame by
                //        set SendPause bit in MACTXCON
                MACTXCON |= SdPause | TxEn ;

                // Step 7. Wait control frame finished
                while ( (BDMASTAT & S_BTxCCP) ) ;
                }

        } while ( (TransferPacket != 'Q') && (TransferPacket != 'q') ) ;
}
```

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 denote 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: Clear BDMA status register bit by write 1.

The BDMA status, every received frame(BRxRDF) bit is cleared by write 1 to BDMASTAT register, so after receive this interrupt , this bit should be cleared.

STEP 3: 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 4: 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 5: Check received frame is valid or not

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

STEP 6: 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 7: Check received frame has error or not

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

STEP 8: Change ownership to BDMA

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

STEP 9: Get next frame descriptor pointer to process

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

STEP 10: Check BDMA NotOwner status bit in the BDMASTAT register

This NotOwner bit means all BDMA frame descriptor is used, so we need set MAC and BDMA control register to receive frame normally.

The source code, for the Ethernet frame receive operation is described in **Listing 3-71**.
In the source code, the pre-defined statement, "KS32C5000_BUG_FETCH" is used for KS32C5000 ethernet controller bug fetch routine. At the end of Ethernet controller description, we describe more detail for the Ethernet controller bug in the KS32C5000.

Listing 3-71. BDMA_Rx_isr() function (MACINIT.C)

```
/*
 * Function : BDMA_Rx_isr
 * Description : Interrupt Service Routine for BDMA Rx
 * Ethenet Frame is received in BDMA_Rx_isr
 */
void BDMA_Rx_isr(void)
{
        sFrameDescriptor *pRxFDptr ;
        U32 RxStatus, FrameLength ;
        U32 CRxPtr;
        U32 sBdmaStat ;
        U8 *pFrameData ;
#if KS32C5000_BUG_FETCH
        U32 FstData,FstDataSave,*FstFrameData ;
#endif

        // Step 1. Get current frame descriptor and status
        CRxPtr = BDMARXPTR ;
        sBdmaStat = BDMASTAT ;

        // Step 2. Clear BDMA status register bit by write 1
        BDMASTAT |= S_BRxRDF ;
        gsBdmaRxStatus.BdmaRxCnt++ ;

        do {
                // Step 3. Check Null List Interrupt
                if ( BDMASTAT & S_BRxNL ) {
                        BDMASTAT |= S_BRxNL ;
                        gsBdmaRxStatus.BRxNLErr++ ;
                        MacInitialize() ;
                        break ;
                }

                // Step 4. Get Rx Frame Descriptor
                pRxFDptr = (sFrameDescriptor *)gCRxFDPtr ;
                RxStatus = (pRxFDptr->StatusAndFrameLength >> 16) & 0xffff;

                // Step 5. If Rx frame is good, then process received frame
                if(RxStatus & RxGood) {
                        FrameLength = pRxFDptr->StatusAndFrameLength & 0xffff ;
```

```
                        PFrameData = (U8 *)pRxFDptr->FrameDataPtr ;
                        GsBdmaRxStatus.BdmaRxGood++ ;
                        if ( !(gsBdmaRxStatus.BdmaRxGood%10000) )
                                Print("<R%d:%d>", tm0.tm_sec,gsBdmaRxStatus.BdmaRxGood) ;

#if KS32C5000_BUG_FETCH
                FstFrameData = (U32 *)pRxFDptr->FrameDataPtr ;
                        FstDataSave = *FstFrameData ;
                        FstData = ( (FstDataSave<<24) & 0xFF000000 ) |\
                                ( (FstDataSave<<8)  & 0x00FF0000 ) |\
                                ( (FstDataSave>>24) & 0x000000FF ) |\
                                ( (FstDataSave>>8)  & 0x0000FF00 ) ;

                        if (gPreviousStatusField == FstData)
                                        pFrameData = pFrameData + 4 ;
#endif

                        // Step 6. Get received frame to memory buffer
                        GetRxFrameData(pFrameData, FrameLength, RxStatus) ;

                } else {
                        // Step 7. If Rx frame has error, then process error frame
                        gErrorPacketCnt++ ;

                        // Save Error status
                        // Check each status, because, error can duplicated
                        if (RxStatus & OvMax) gsMacRxStatus.OvMaxSize++ ;
                        if (RxStatus & CtlRecd) gsMacRxStatus.sCtlRecd++ ;
                        if (RxStatus & Rx10Stat) gsMacRxStatus.sRx10Stat++ ;
                        if (RxStatus & AlignErr) gsMacRxStatus.AllgnErr++ ;
                        if (RxStatus & CRCErr) gsMacRxStatus.sCRCErr++ ;
                        if (RxStatus & Overflow) gsMacRxStatus.OverflowErr++ ;
                        if (RxStatus & LongErr)   gsMacRxStatus.sLongErr++ ;
                        if (RxStatus & RxPar) gsMacRxStatus.RxParErr++ ;
                        if (RxStatus & RxHalted) gsMacRxStatus.sRxHalted++ ;
                }

                // Step 8. Change ownership to BDMA for next use
                (pRxFDptr->FrameDataPtr) |= fOwnership_BDMA;

                // Save Current Status and Frame Length field, and clear
#if KS32C5000_BUG_FETCH
                gPreviousStatusField = pRxFDptr->StatusAndFrameLength ;
#endif
                pRxFDptr->StatusAndFrameLength = (U32)0x0;

                // Step 9. Get Next Frame Descriptor pointer to process
                gCRxFDPtr = (U32)(pRxFDptr->NextFrameDescriptor) ;

        } while (CRxPtr != gCRxFDPtr);

        // Step 10. Check Notowner status
        if ( sBdmaStat & S_BRxNO ) {
```

```
                BDMASTAT |= S_BRxNO ;
                GsBdmaRxStatus.BRxNOErr++ ;

                ReadyMacRx() ;
        }

        BdmaRxDoneFlagForLoopBackCheck = 1 ; // only used for loopback test
}
```

The receive operation in the BDMA Rx interrupt service routine is described in **Figure 3-35**.

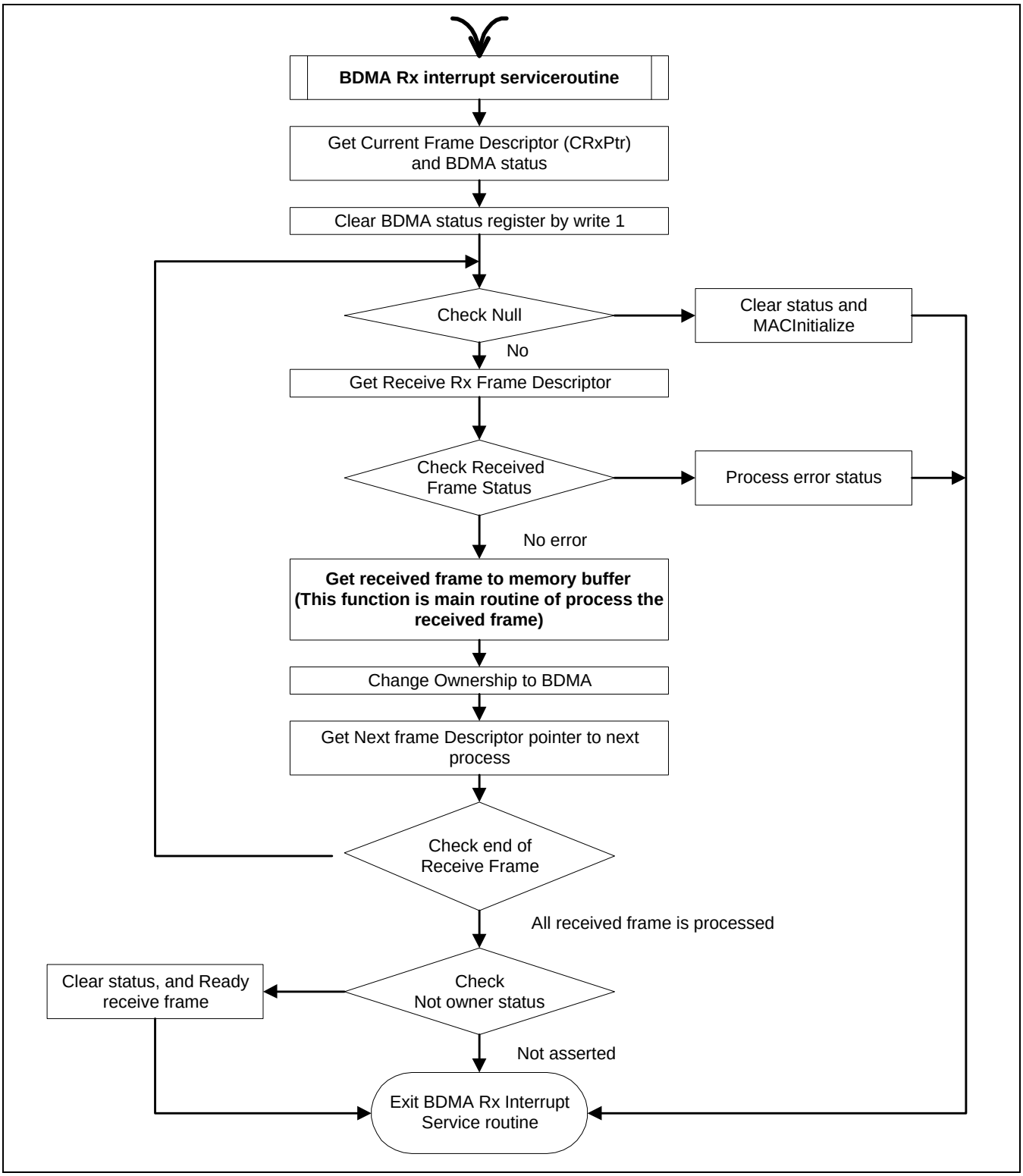


Figure 3-35. Ethernet Frame Reception Flow

KS32C5000 Ethernet controller bug fetch

KS32C5000 has two bug in the Ethernet controller. First one is store invalid word in the front of receive frame buffer at the special frame length, this frame length is changed by BDMA receive burst size. The second thing is MAC controller is hanged after receive heavy short frame from 100 Mbps network.

The solution for first bug, you can see the pre-defined statement in the BDMA_Rx_isr() function. The invalid word in the first of receive frame buffer is same as word swapped value of previous frame descriptor' s status and length field. So, we can check the first word with previous frame descriptor' s status and length value. If invalid word is attached, we can move the frame buffer pointer to next 4 byte. The source code is listed in **Listing 3-72**.

Listing 3-72. KS32C5000 bug fetch code 1 (MACINIT.C)

```
.
.
#if KS32C5000_BUG_FETCH
        U32 FstData,FstDataSave,*FstFrameData ;
#endif
.
.
.
#if KS32C5000_BUG_FETCH
                FstFrameData = (U32 *)pRxFDptr->FrameDataPtr ;
                        FstDataSave = *FstFrameData ;
                        FstData = ( (FstDataSave<<24) & 0xFF000000 ) |\
                                ( (FstDataSave<<8)  & 0x00FF0000 ) |\
                                ( (FstDataSave>>24) & 0x000000FF ) |\
                                ( (FstDataSave>>8)  & 0x0000FF00 ) ;

                        if (gPreviousStatusField == FstData)
                                pFrameData = pFrameData + 4 ;
#endif
.
.
.
#if KS32C5000_BUG_FETCH
                gPreviousStatusField = pRxFDptr->StatusAndFrameLength ;
#endif
.
.
```

The solution for second bug is use a kind of watch-dog timer, in the every system has system tick timer, so we can use this timer to call Ethernet watch-dog timer. In the watch-dog timer, check the receive frame descriptor pointer, and there is no change in the several times, then initialize Ethernet controller receive function. The watch-dog timer for Ethernet controller and initialize function is described in **Listing 3-73**.

Listing 3-73. KS32C5000 bug fetch code 2 (MACINIT.C)

```
/*
 * Function : Lan_WatchDog
 * Description : LAN watchdog function
 */
void Lan_WatchDog(void)
{
	WatchDogTime++ ;
	if ( !(WatchDogTime%100) ) {
		if (gCRxFDPtr == gPreviousFdp) {
				MacWatchdogInit() ;
			}
		else gPreviousFdp = gCRxFDPtr ;
	    }
}

// Initialize function for WatchDog
void MacWatchdogInit(void)
{
	sFrameDescriptor *pRxFDptr ;

	BDMARXCON = 0x0 ;		// Disable BDMA contoller Rx operation
	BDMARXCON = BRxRS ;		// Reset BDMAC Receiver
	Disable_Int(nBDMA_RX_INT) ;	// Disable BDMA Rx Interrupt
	BDMARXLSZ = MaxRxFrameSize ;	// 1520
	pRxFDptr = (sFrameDescriptor *)gCRxFDPtr ;
	BDMARXPTR = gCRxFDPtr = (U32)(pRxFDptr->NextFrameDescriptor) ;
	Enable_Int(nBDMA_RX_INT);
	BDMARXCON = gBdmaRxCon ;
}
```

The sample TIMER interrupt service routine for call the watch-dog timer is described in **Listing 3-74**. This interrupt service routine can be changed by system usage.

Listing 3-74. Tm0isr function for KS32C5000 bug fetch code 2 (TIMER.C)

```
/* timer0 can be used for system real time clock */
void tm0isr(void)
{
        clk_tick0++;
        IOPDATA = ~(1<<tm0.tm_sec%4);
        Lan_WatchDog() ;
        if(clk_tick0 == TICKS_PER_SECOND) {
                clk_tick0 = 0;
                tm0.tm_sec++ ;
        }
}
```

NOTES

4 SYSTEM DESIGN

OVERVIEW

The KS32C5000(A)/50100, SNASUMG's 16/32-bit RISC microcontroller is cost-effective and high performance microcontroller solution for Ethernet-based system. The integrated on-chip functions of KS32C5000(A)/50100 are

- 8K-byte unified cache/SRAM
- Ethernet controller
- HDLC controller
- UART
- Timer
- I2C
- Programable I/O ports
- Interrupt controller

Therefore, you can use KS32C5000(A)/50100 as amount types of system.

APPLICABLE SYTEM WITH KS32C5000(A)50100

If your product need to be networked, the KS32C5000(A)/50100, SNASUMG's 16/32-bit RISC microcontroller can be reduce your system cost. There are sample system, it can be designed with KS32C5000(A)/50100.

- Managed hub/Switch
- Router and bridge
- ISDN Router /TA
- ADSL Router
- Home/Industry security
- Cable modem
- Digital camera
- Home/Industry security
- Cable modem
- Internet FAX/Internet phone
- Network printer
- Game machine
- Any system with network interface

MEORY INTERFACE DESIGN

ADDRESS BUS GENERATION

The KS32C5000(A)/50100 address bus generation is based on the required data bus width of each memory bank, The internal system address bus is shifted out to an external address bus, ADDR[21:0]. This means that memory control signals such as nRAS[3:0], nCAS[3:0], nECS[3:0],nRCS[5:0], nWBE[3:0] are generated by the system manager according to a pre-configured external memory scheme (see Table 4-1,and Figure 4-1).

Table 4-1. Address Bus Generation Guidlines

Data Bus Width	External Address Pins, ADDR[21:0]	Accessible Memory Size
8-bit (Byte)	A21-A0 (internal)	4M bytes
16-bit (Half-Word)	A22-A1 (internal)	4M half-words
32-bit (Word)	A23-A2 (internal)	4M words

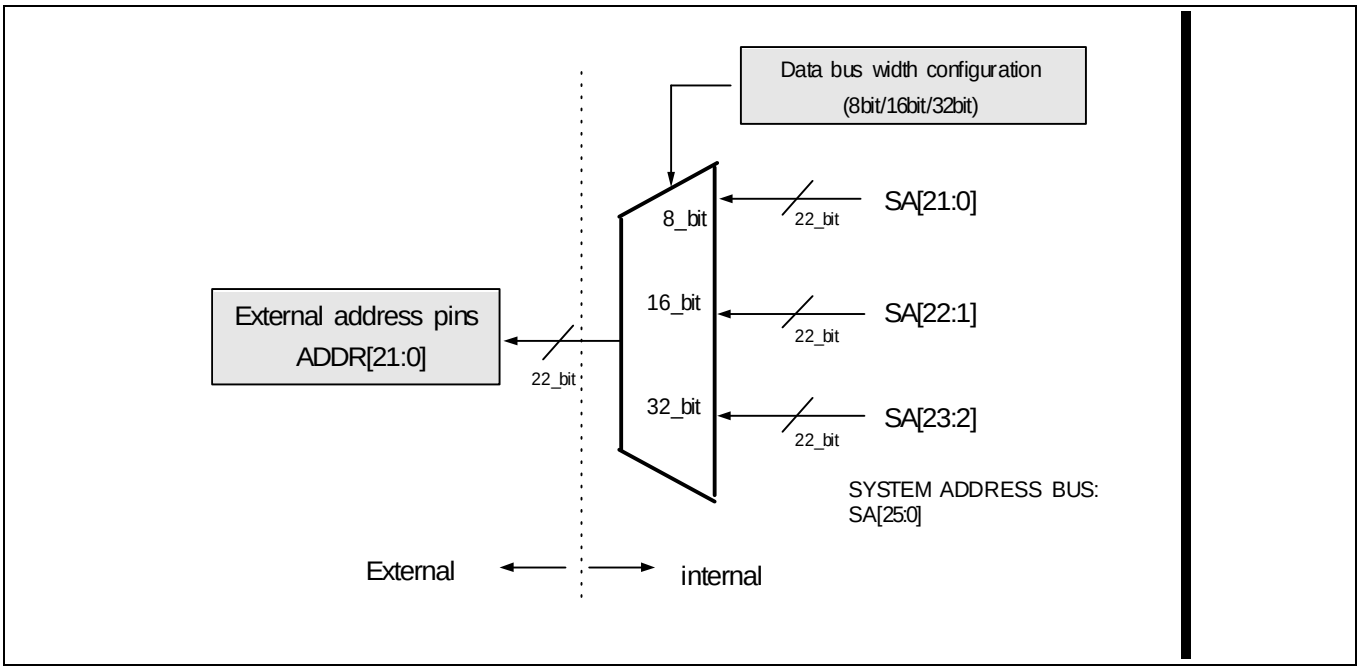


Figure 4-1. External address bus generation(ADDR[21:0])

BOOT ROM DESIGN

When system reset, a KS32C5000(A)/50100A access 0x00000000 address. And KS32C5000(A)/50100 should be configure some system variable after reset. Therefore this special code (BOOT ROM image) should be located on address 0x00000000. A boot ROM can have a various width of data bus, and it is controlled by B0SIZE[1:0] pins.

Table 4-2. Data Bus Width for ROM Bank 0

B0SIZE[1:0]	Data Bus Width
00	Reserved
01	8-bit (byte)
10	16-bit (half-word)
11	32-bit (word)

ONE BYTE BOOT ROM DESIGN

A design with one byte boot ROM is shown in Figure 4-2.

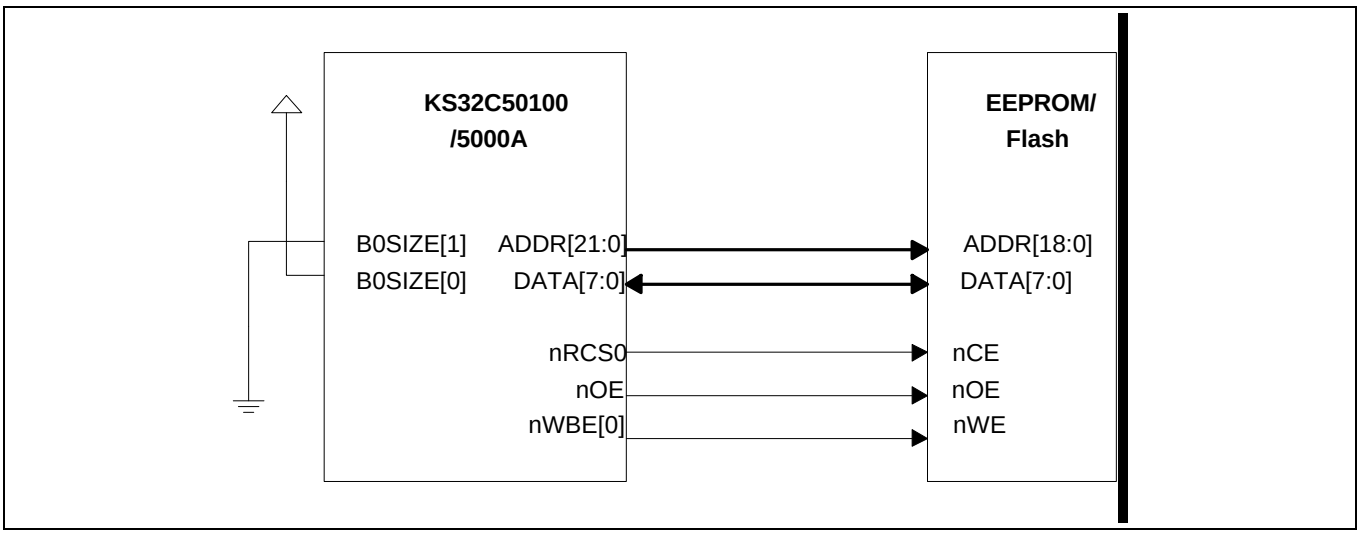


Figure 4-2. One Byte Boot ROM Design

MAKE AND FUSING ONE BYTE ROM IMAGE

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

HALF-WORD BOOT ROM DESIGN WITH BYTE EEPROM/FLASH

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

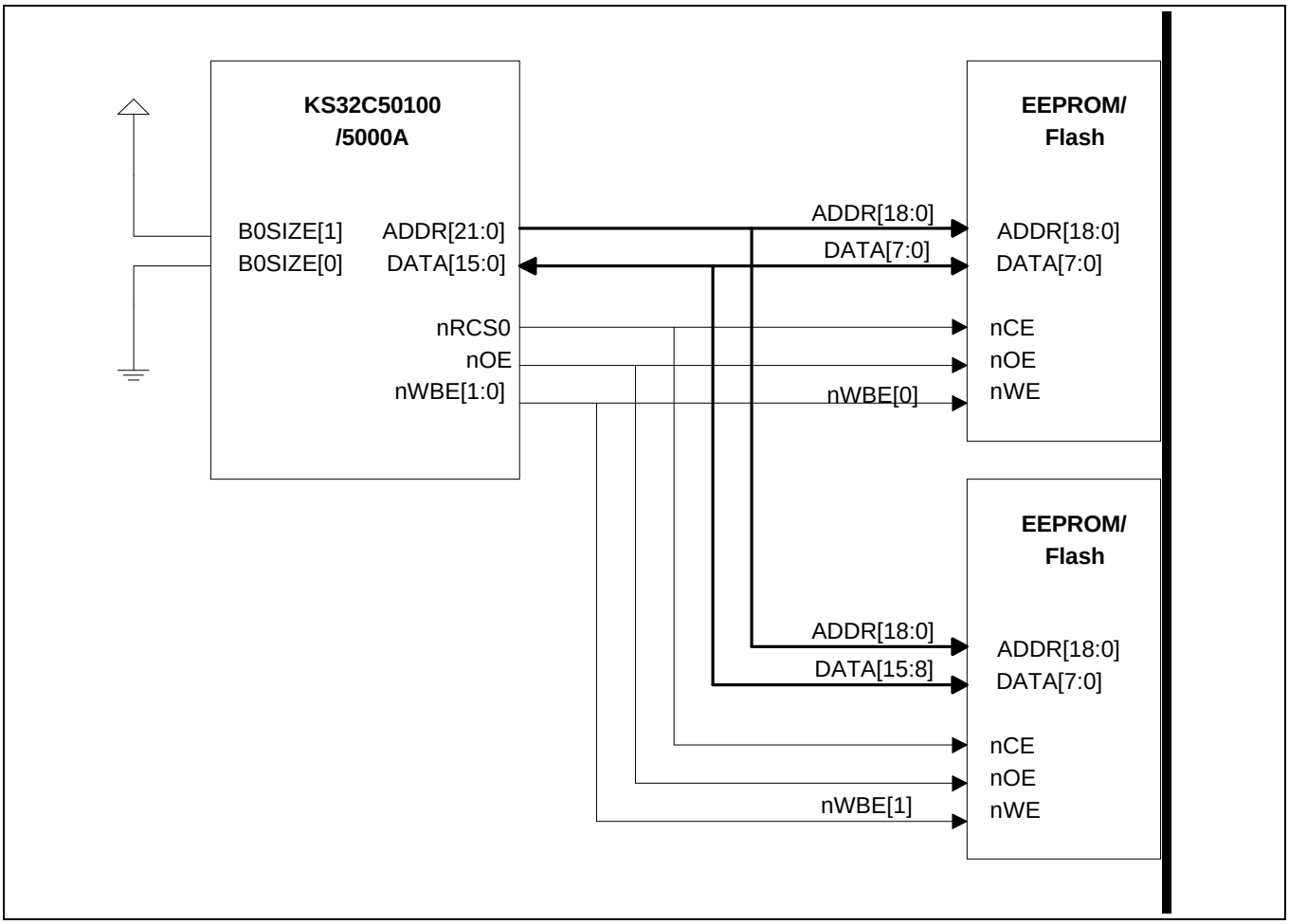


Figure 4-3. The Half-Word Boot ROM Design with Byte EEPROM/Flash

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

When make half-word ROM image, you can split two image files, EVEN and ODD. When you use our device KS32C5000(A)/50100 as big-endian mode, then you should swap EVEN and ODD. Because KS32C5000(A)/50100 has little-endian data bus structure.

For example :

split2 rom.bin
Output even file name : rom.ODD
Out odd file name : rom.EVN

But, when you use our device as little-endian mode, then there is no need swap of EVEN and ODD.

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

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

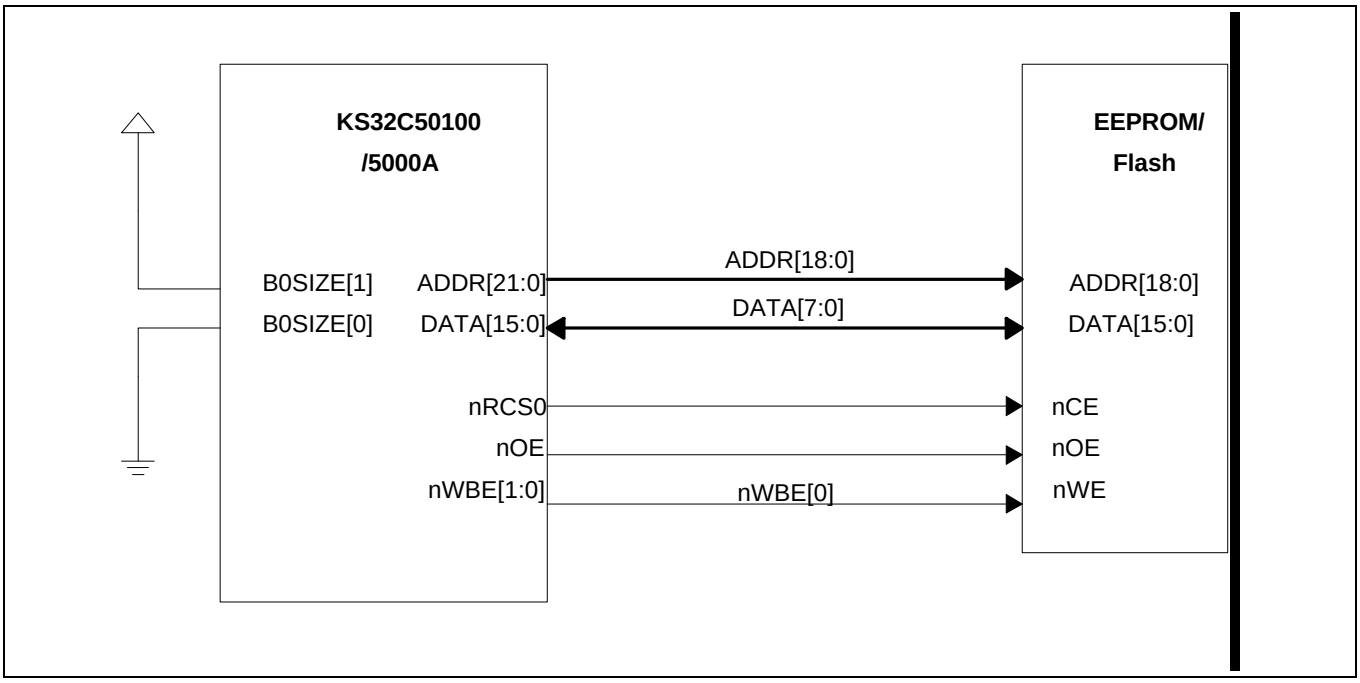


Figure 4-4. The Half-Word Boot ROM Design with Half-Word EEPROM/Flash

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

When make half-word ROM image, you should swap EVEN and ODD ROM image on ROM writer. Because KS32C5000(A)/50100 has little-endian data bus structure.

But, when you use our device as little-endian mode, then there is no need swap of EVEN and ODD.

MEMORY BANKS DESIGN AND CONTROL

The KS32C50100 has 6 bank ROM/SRAM banks (ROM0 bank for boot ROM), 4 EDO/Synchronous DRAM banks, and 4 external I/O banks,. The KS32C5000(A) has 6 bank ROM/SRAM banks(ROM0 bank for boot ROM), 4 EDO DRAM banks, and 4 external I/O banks, The system manager on KS32C5000(A)/50100 can control access time, data bus width, and base/end point, for each banks by S/W. The access time and base/end point of ROM/SRAM banks is controlled by ROMCON0-5 control register on system manager. The base point of external I/O banks is controlled by REFEXTCON control register, and access time for each external I/O banks is controlled by EXTCON0-1 control register. And the access time and base/end point of DRAM banks is controlled by DRAMCON0-3 control register.

The care is need for attach the memory and external I/O to KS32C5000(A)/50100's, Even though CPU is configured for the big-endian memory accessing. However, to connect with the external memory, KS32C5000(A)/50100 is different with the other normal big-endian microcontrollers because of its internal byte-swap mechanism. The data connection with external memory should be done as the "little endian" way. That is, the byte 0 of the external memory(that is, the most significant byte of a word)should be connected to the controller's data pin[7:0], byte 1 to data pin[15:8], byte 2 to data pin[23:16], and byte 3 to data pin[31:24].

When you use KS32C50100 with SDRAM, you can enable SDRAM mode by SYSCFG register bit 31.

The data bus width for each ROM/SRAM banks, external I/O banks, and DRAM banks is controlled by EXTDBWTH data bus width register on system manager, except ROM bank 0.

The ROM bank 0 is used for boot ROM bank, therefore bank 0 is controlled by H/W, B0SIZE[1:0] is used for this purpose(see table 3-2).

The control of EXTDBWTH,ROMCON0-5, DRAMCON0-4,REFEXTCON is performed when system reset, by special command, LDMIA and STMIA. sample code for special register configuration is described below.

ROM/SRAM BANKS DESIGN

The ROM/SRAM banks 1-5, can have a various width of data bus, and the bus width is controlled by S/W, A EXTDBWTH special register set. A sample design for ROM/SRAM bank 1-5 is shown in Figure 4-5, Figure 4-6, and Figure 4-7.

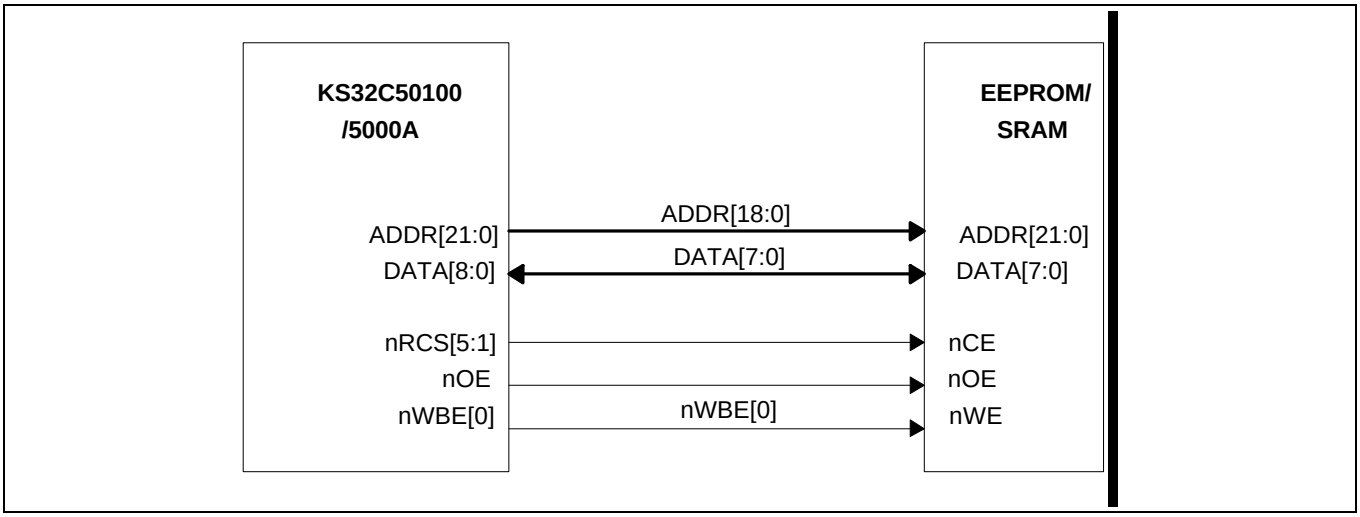


Figure 4-5. One-byte EEPROM/SRAM Banks Design

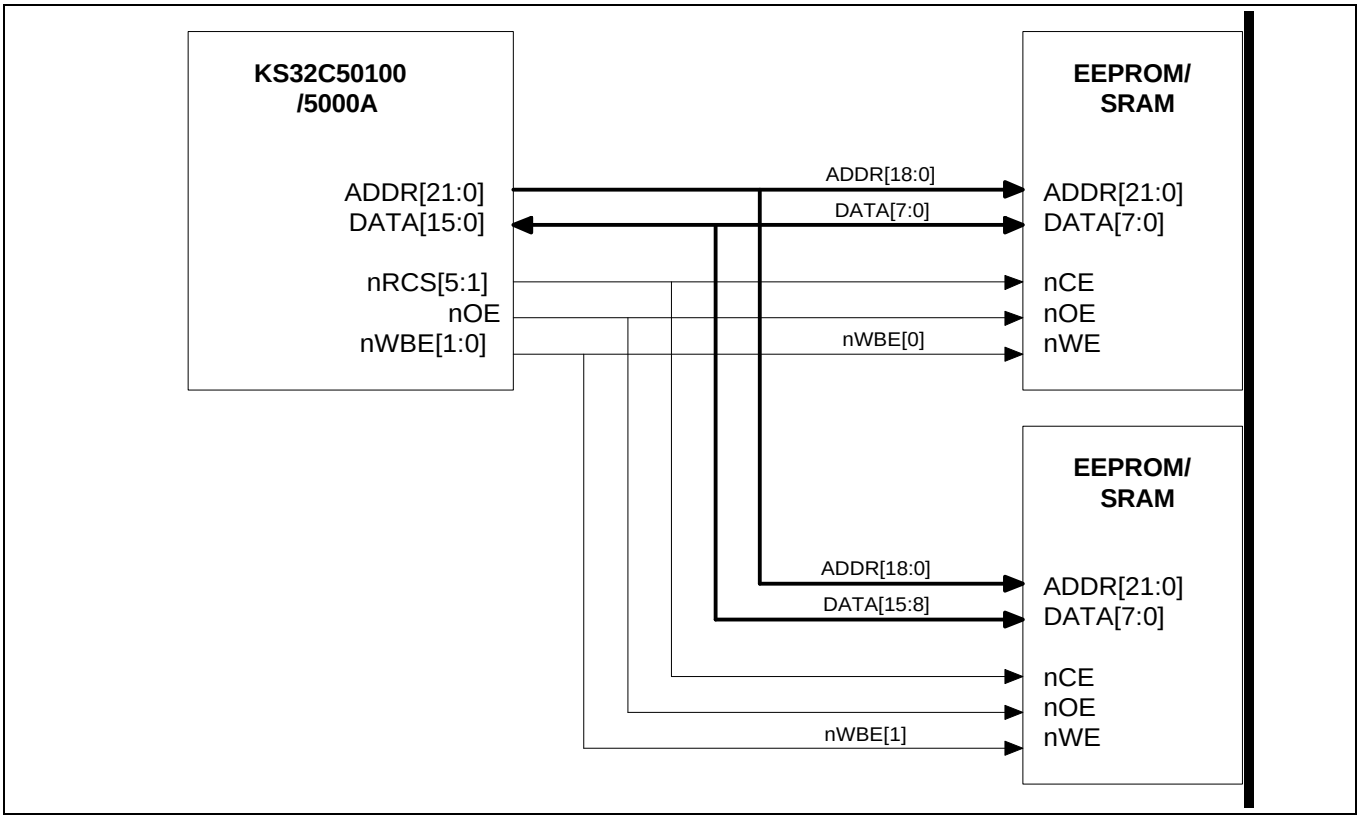


Figure 4-6. Half-word EEPROM/SRAM Banks Design

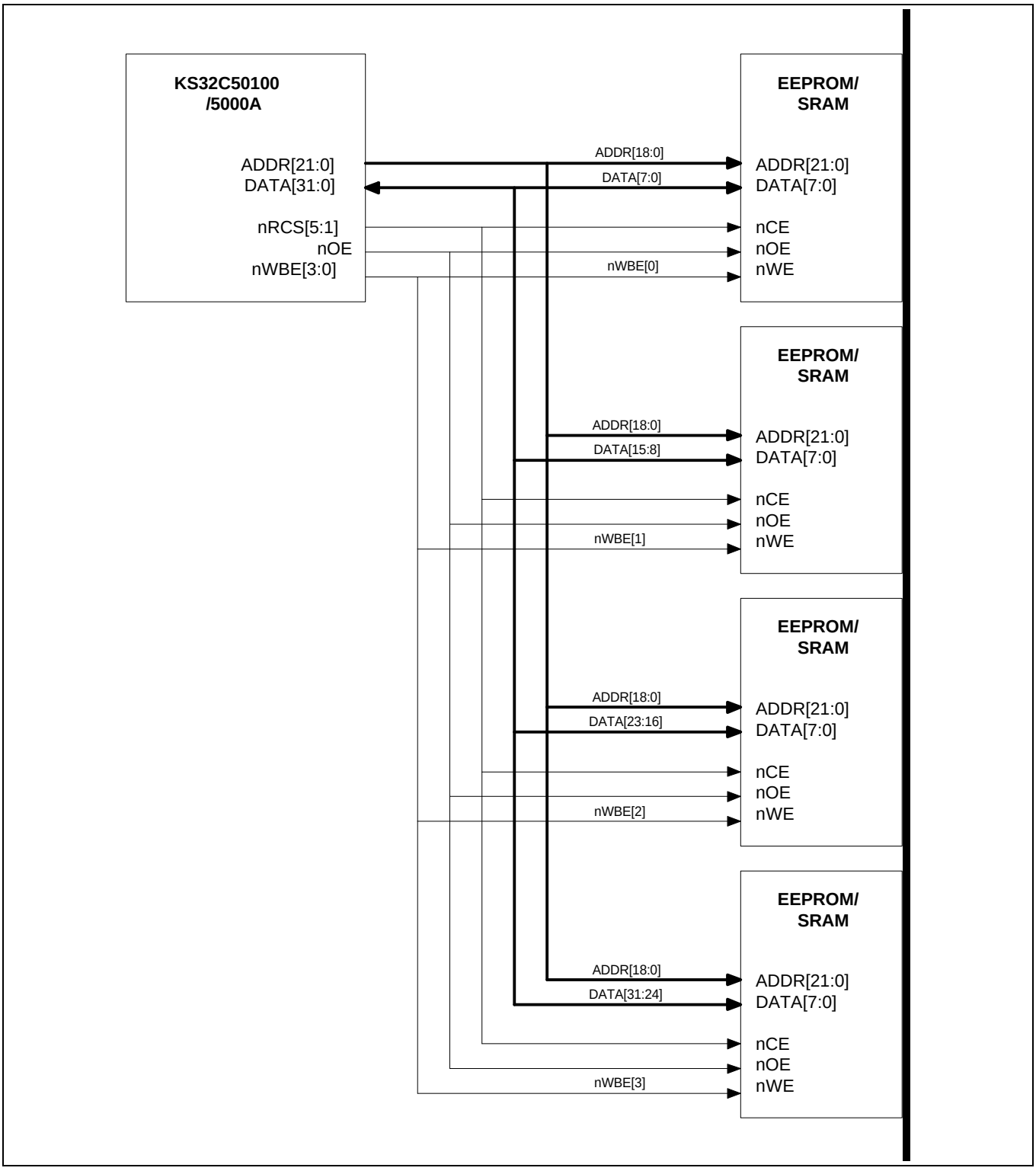


Figure 4-7. Word EEPROM/SRAM banks design

ROM/SRAM BANK 5 DESIGN WITH MULTIPLEXED ADDRESS/DATA BUS

The KS32C5000(A)/50100 supports multiplexed address/data bus for low-cost chips which require multiplexed bus. But, ROM Bank5 muxed bus mode was added to KS32C5000(A)/50100 for the purpose of using special ASIC chip interface not commercial device. So, there are some limits using this mode, The address bus generation scheme of KS32C5000(A)/50100 is shifted address bus according to the data bus width of each memory bank.

The ROM bank 5 restriction is listed in below.

- Data bus width is fixed to 32bit. So, if you want to attach 8-bit device, then you should read all 32-bit data and then process by S/W to get 8-bit real data.
- Multiplexed address signal has same configuration with stand-alone address bus, so it can make exhaust accessable addressing area than real address area. That is, If you configure the data bus width of ROM Bank5 to word in muxed bus mode, then the muxed address will be appeared on the data bus shifted by two bits. Using this mode in spite of the loss address space less then 64Mbyte, you have to use trick as follows

  (ROM Bank5 offset address + access address << n) + byte offset.

  Where, n is 2 in case of 32bit data bus width, 1 in case of 16bits, 0 in case of 8bits. byte offset have to be 0 for byte access, 1 for halfword access, 2 for word access. For example, you want to access half-word 0x123 address, translated address is as follows:

  0x123 << 2 + 3 = 0x48f
- You can use the MCLKO clock output as a target device.

EDO DRAM BANKS DESIGN FOR KS32C5000(A)/50100

The DRAM banks 0-4, can have a various width of data bus, and the bus width is controlled by S/W, A EXTDBWTH special register set. A sample design for DRAM bank 0-3 is shown in Figure 4-8, Figure 4-9, and Figure 4-10.

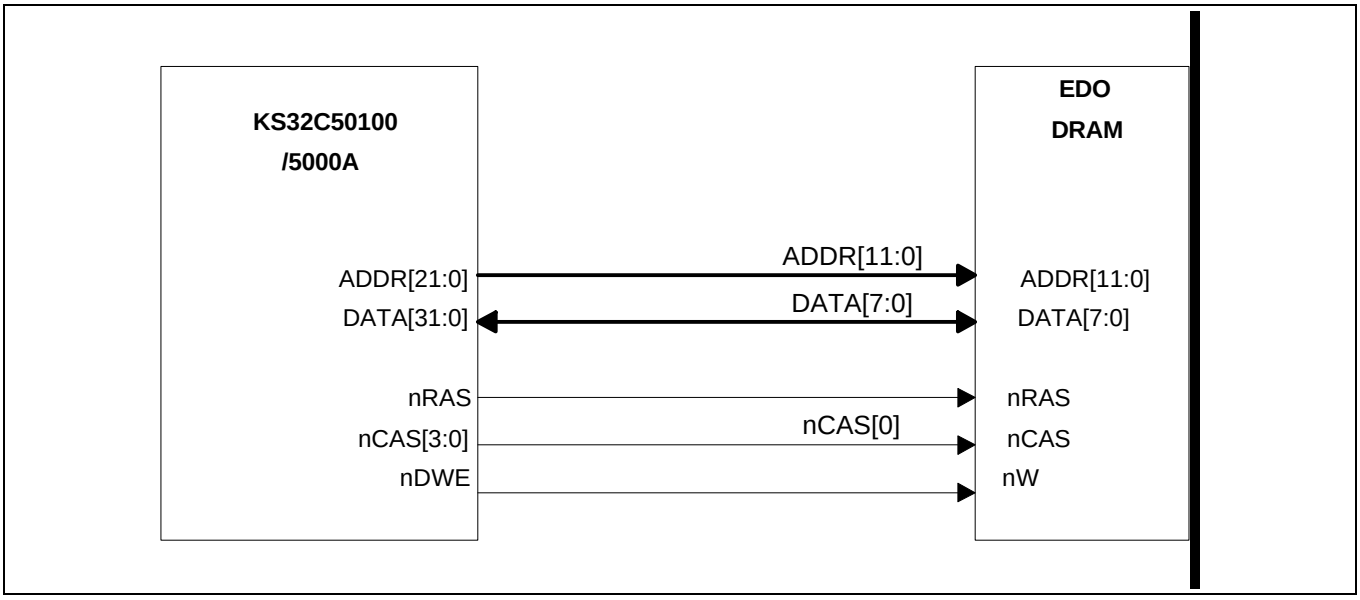


Figure 4-8. 1-byte EDO/Normal DRAM Banks Design

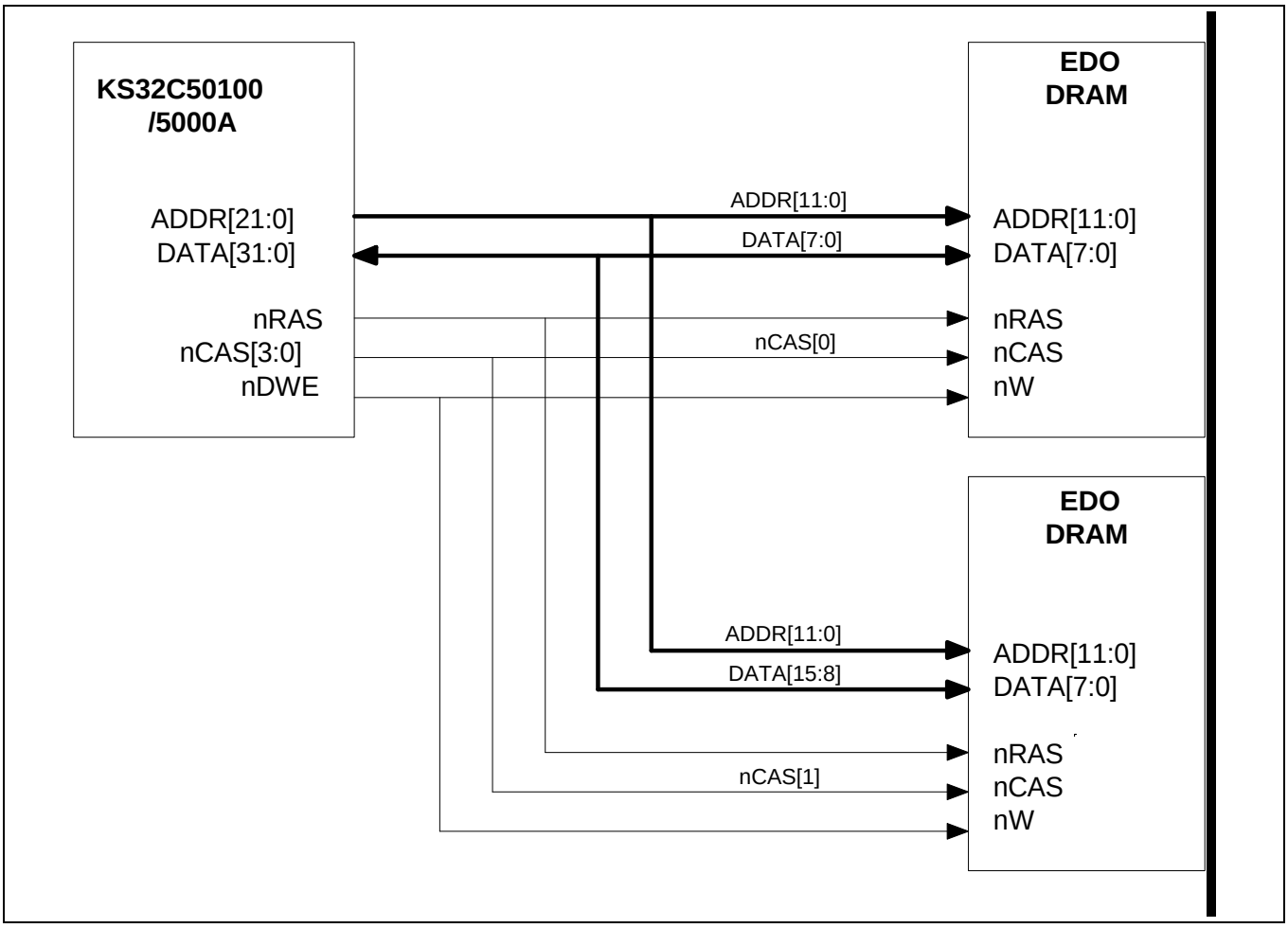


Figure 4-9. Half-word EDO/Normal DRAM Banks Design

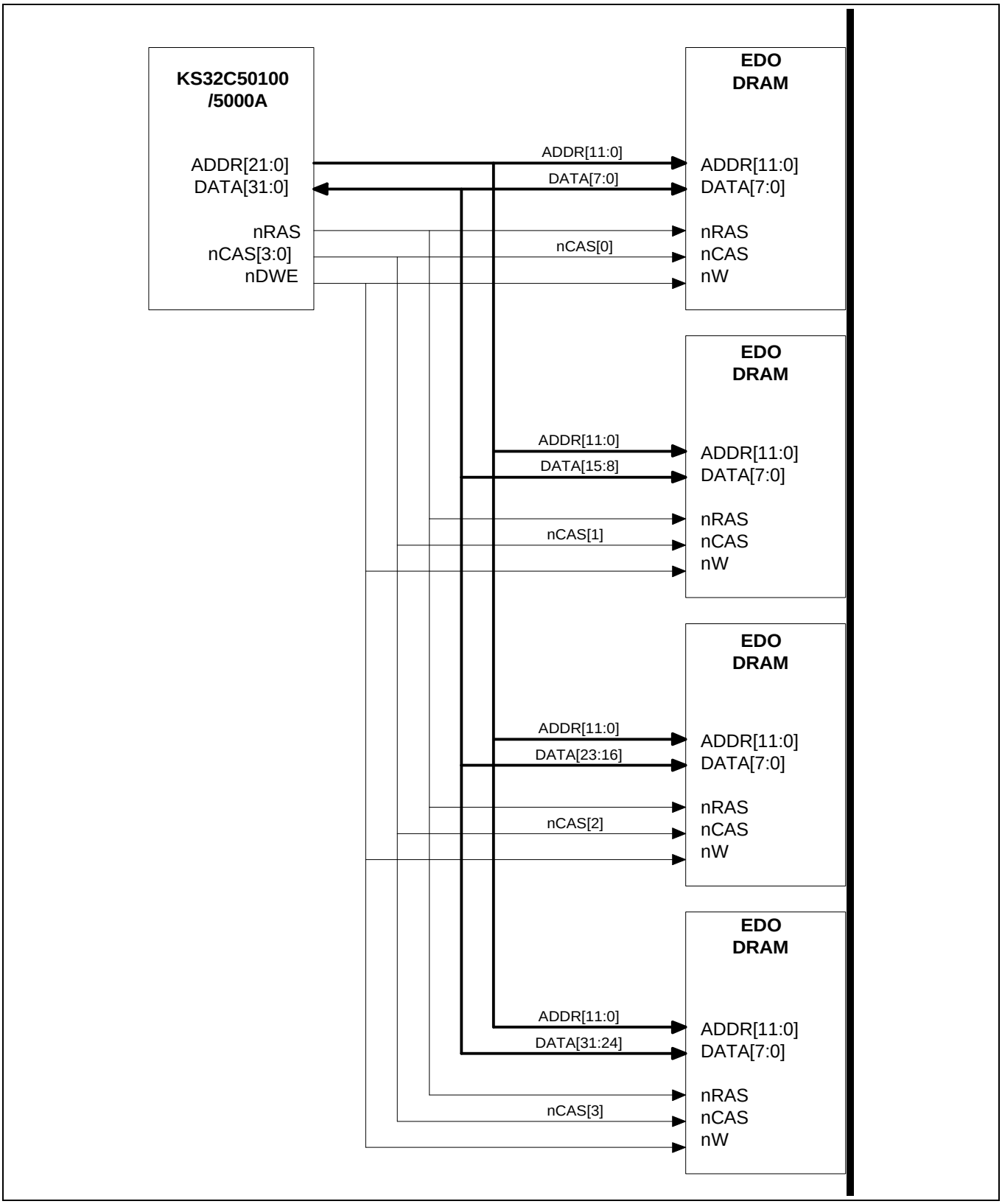


Figure 4-10. Word EDO/Normal DRAM Banks Design

SYNCHRONOUS DRAM BANKS DESIGN FOR KS32C50100

The KS32C50100 Synchronous DRAM interface features are as follows :

- Maximum column address of SDRAM: 11bit
- CAS latency: 2 cycle
- Burst lengyh support: only 1 burst mode(single mode) support
- Burst type: sequential
- Support auto refresh

KS32C50100 can support below Samsung' s SDRAM configuration for each bank.

- 16M SDRAM
  - ✓ KM44S4020C: 2M x 4bit with 2 banks
  - ✓ KM48S2020C: 1M x 8bit with 2 banks
  - ✓ KM416S1020C: 512K x 16bit x 2 banks
- 64M 2 banks SDRAM
  - ✓ KM44S16020B: 8M x 4bit with 2 banks
  - ✓ KM48S8020B: 4M x 8bit with 2 banks
  - ✓ KM416S4020B: 2M x 16bit x 2 banks
- 64M 4 banks SDRAM
  - ✓ KM44S16030B/C: 4M x 4bit with 4 banks
  - ✓ KM48S8030B/C: 2M x 8bit with 4 banks
  - ✓ KM416S4030B/C: 1M x 16bit x 4 banks
- 2Mx32 4 banks SDRAM
  - ✓ KM432S2030B: 512K x 32bit with 4 banks

When you desgn with SDRAM, you should enable MCLKO pin for SDRAM sync. Clock. And Address 10 is used for bank address (BA) select address for SDRAM.

The required SDRAM interface pin is CKE, SDCLK, nSDCS[3:0], nSDCAS, nSDRAS, DQM[3:0], ADDR[10]/AP. The sample design with SDRAM is shown in Figure 4-11, and Figure 4-12.

When you design with 5 V device with 3.3 V SDRAM, then you need to protect the 3.3 V SDRAM from 5 V device. That is, data bus should be has 10 ohm damping resister from 5 V device.

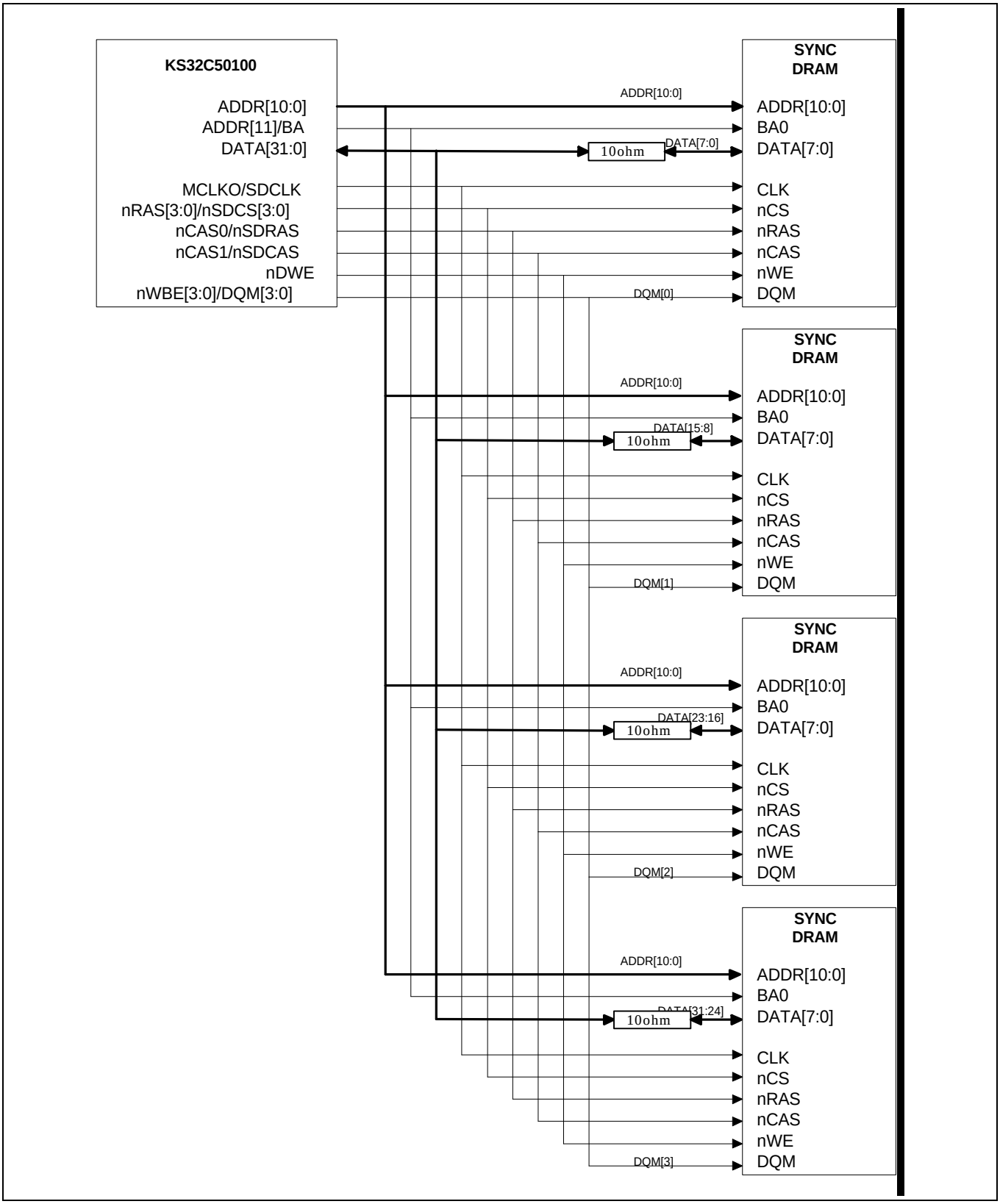


Figure 4-11. SDRAM Design with 1-byte Components

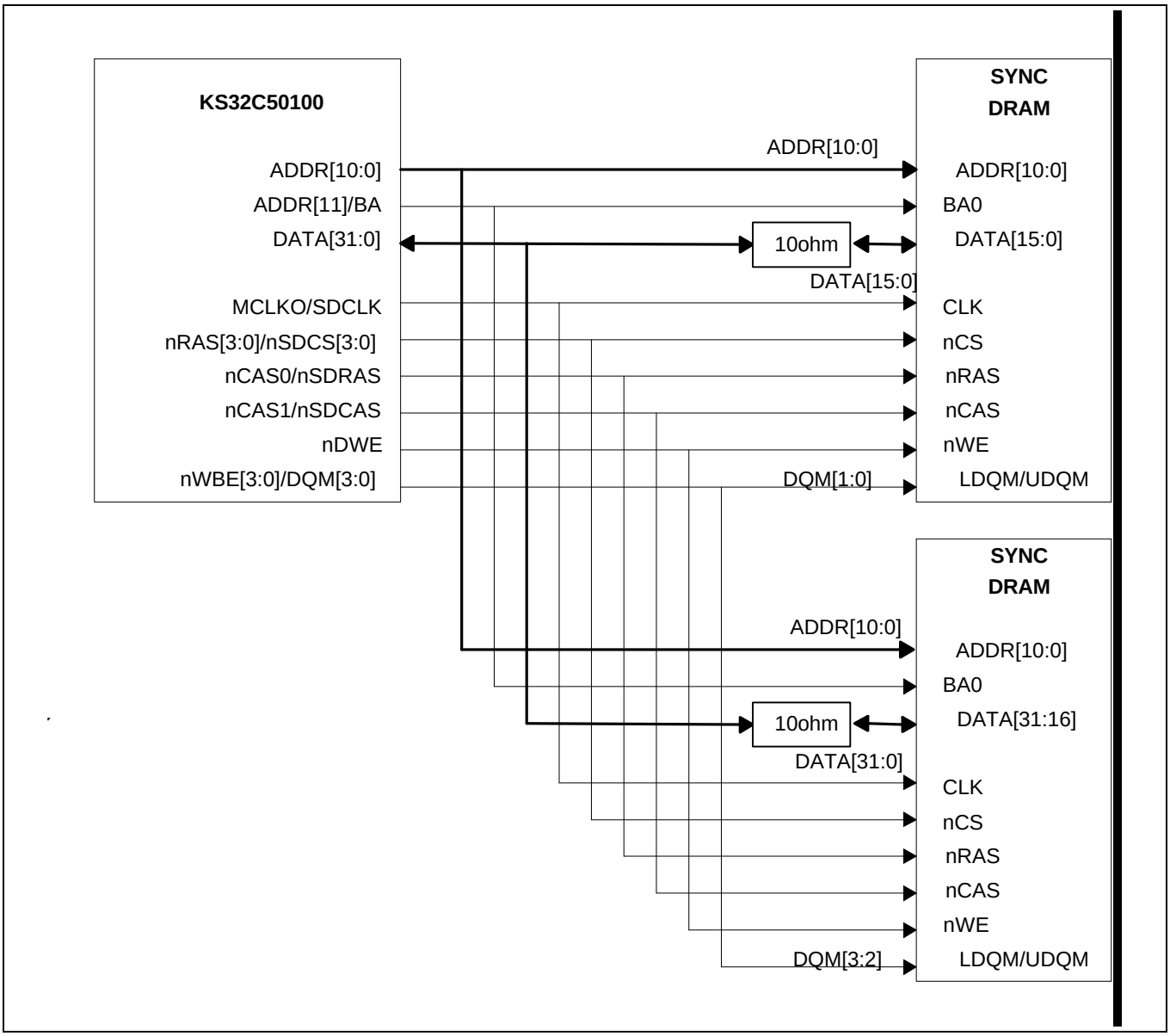


Figure 4-12. SDRAM Design with Half-word Components

EXTERNAL I/O BANKS DESIGN

The external I/O banks 0-3, can have a various width of data bus, and the bus width is controlled by S/W, A EXTDBWTH special register set. A design with external I/O banks 0-3 to memory-like device is similar to ROM/SRAM banks 1-5.

For a very slow device, the KS32C5000(A)/50100 provide external wait request signal (nEWAIT). To use this request signal, you set the tCOS and tCOH should not be zero. and nEWAIT signal should be assert, as soon as possible, the nECS chip select signal is asserted. For detail timing for use of nEWAIT, you can see KS32C5000(A)/50100 USER'S MANUAL, System Manager block description.

The sample design timing diagram with nECS and nEWAIT is deficted in below Figure 4-13.

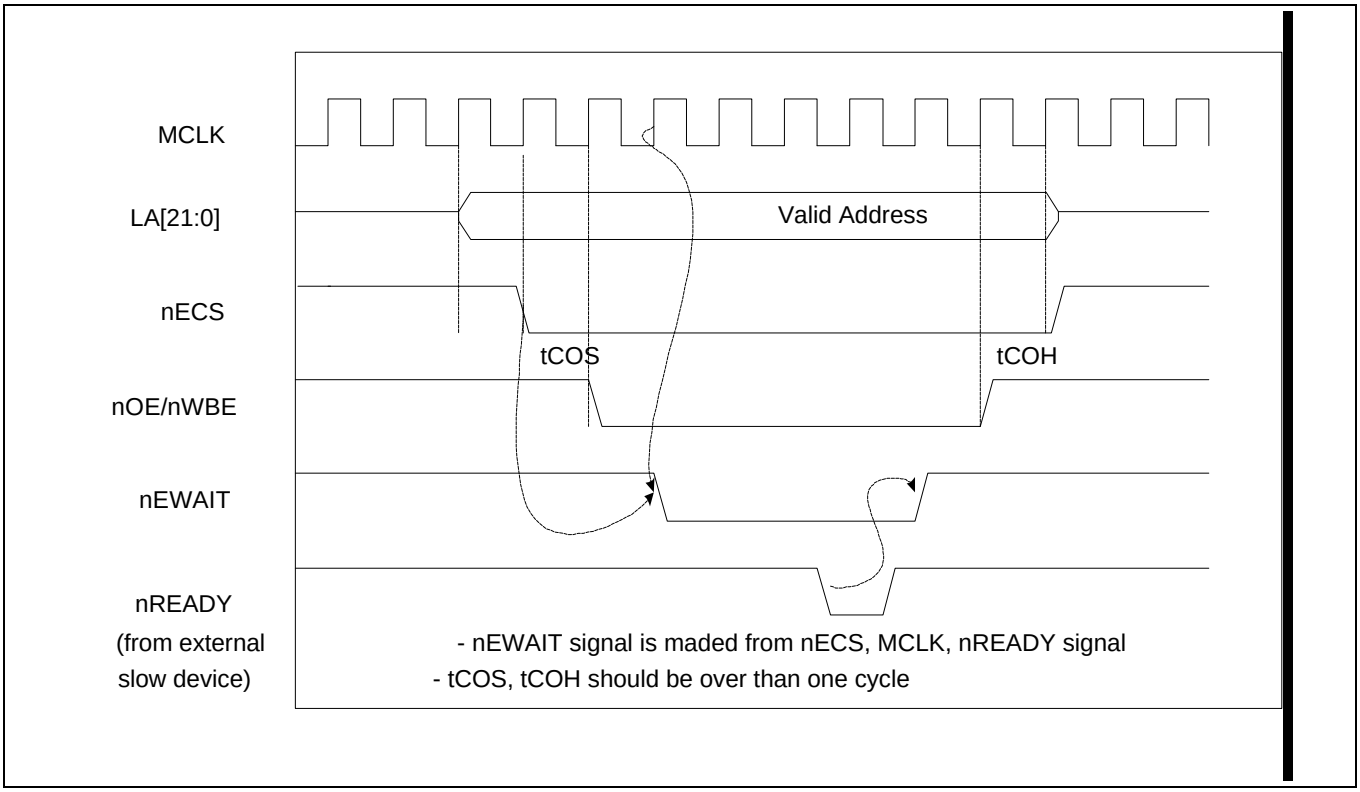


Figure 4-13. External Slow Device Design with nEWAIT Signal

1. Sample Design with External I/O banks: General UART chipset interface

In this design, we use external I/O bank 0, and one byte data bus width.

The external I/O access cycle should be configured as proper value, by software.

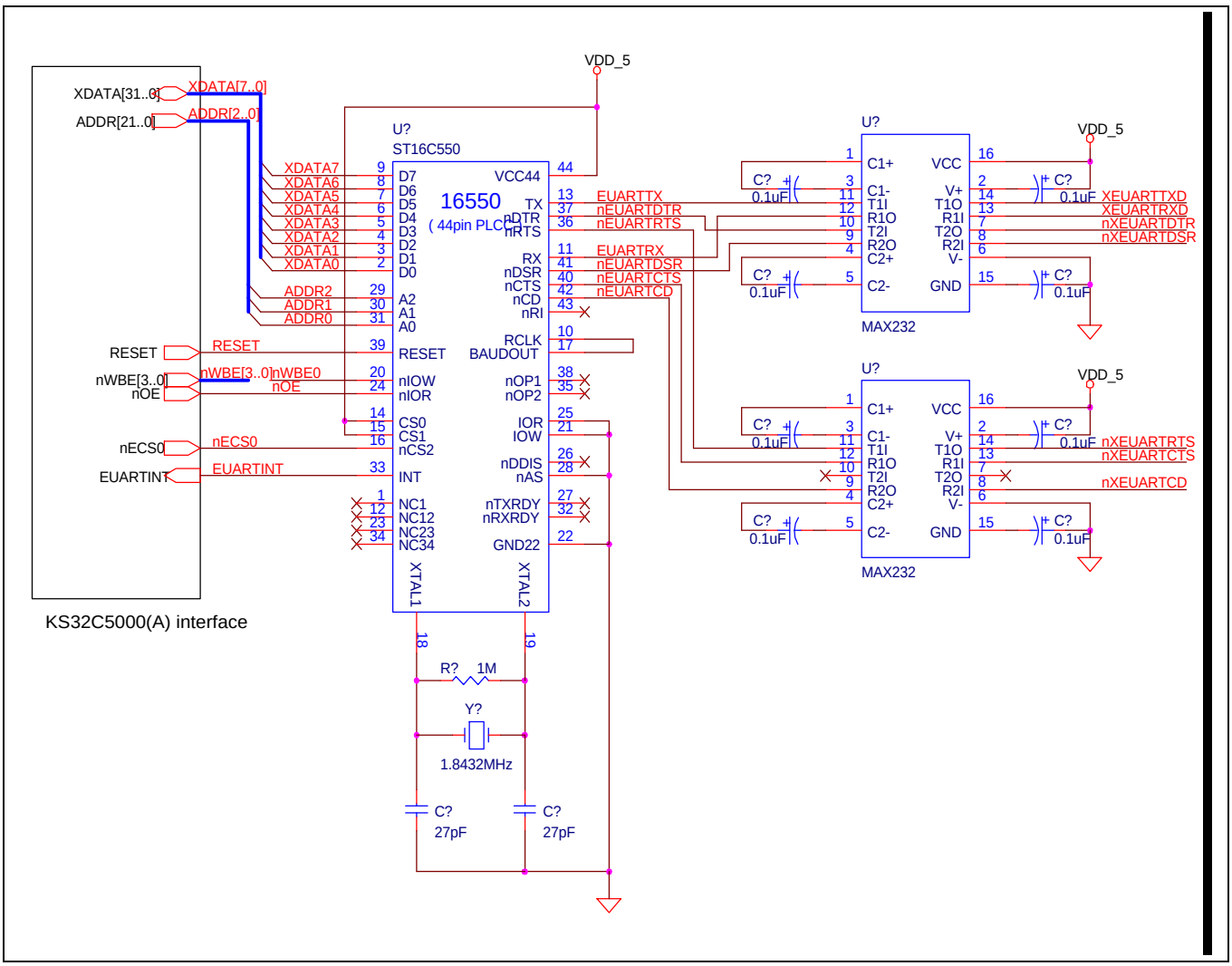


Figure 4-14. External UART Chipset Interface Design

2. Sample Design with External I/O banks: Super-I/O

In this design, we use external I/O bank 0, and one byte data bus width.

The external I/O access cycle should be configured as proper value, by software. And you can refer datasheet of Super-I/O device for more detail operation.

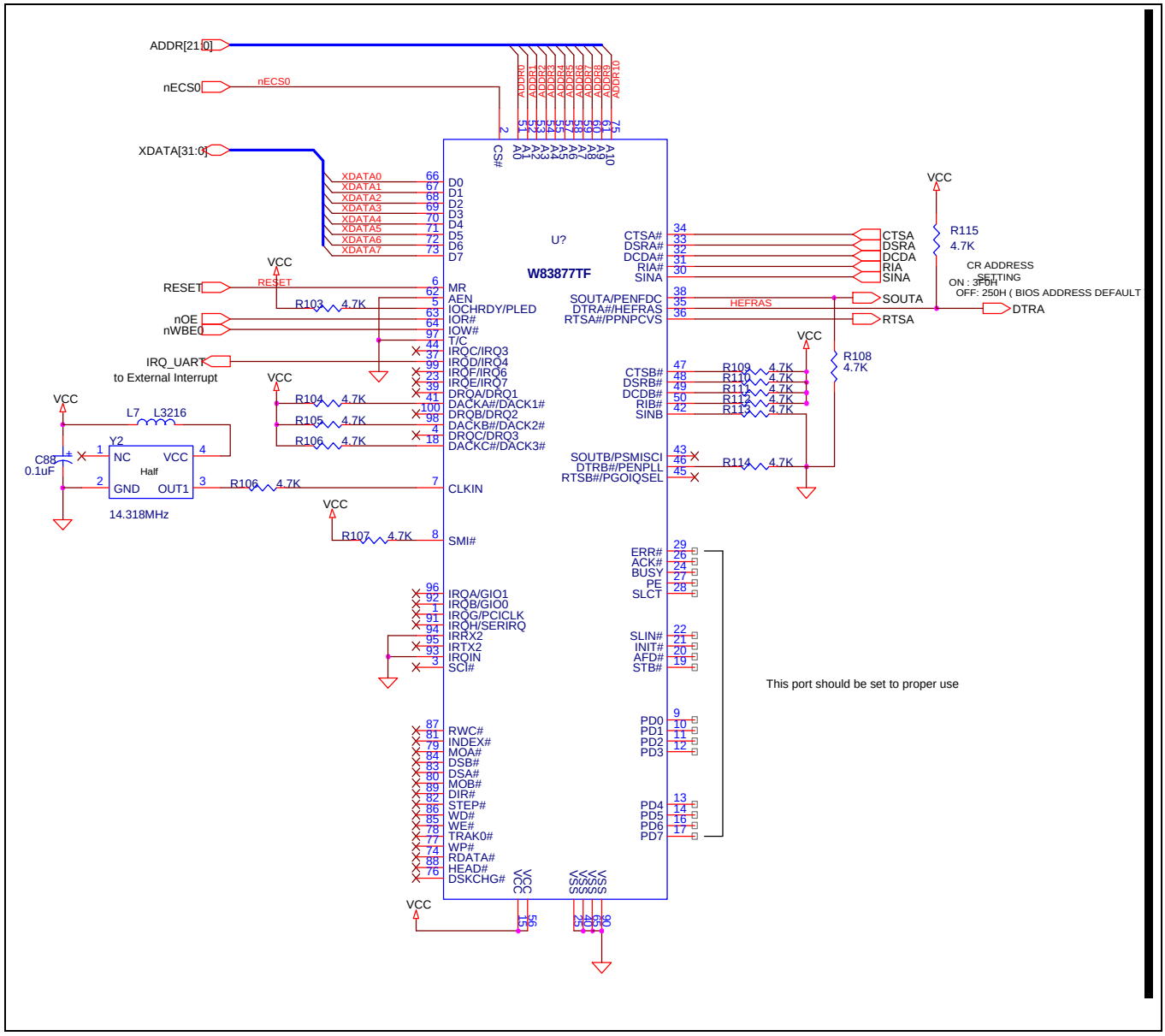


Figure 4-15. Super-I/O Interface Design

3. Sample Design with External I/O banks: ISA bus interface design

The ISA bus has 16 bit data bus width and I/O, memory operation with 8MHz master clock, so we need some kinds of EPLD or CPLD for design ISA bus interface. The access cycle and data bus width should be setted to proper value for generate ISA interface signal.

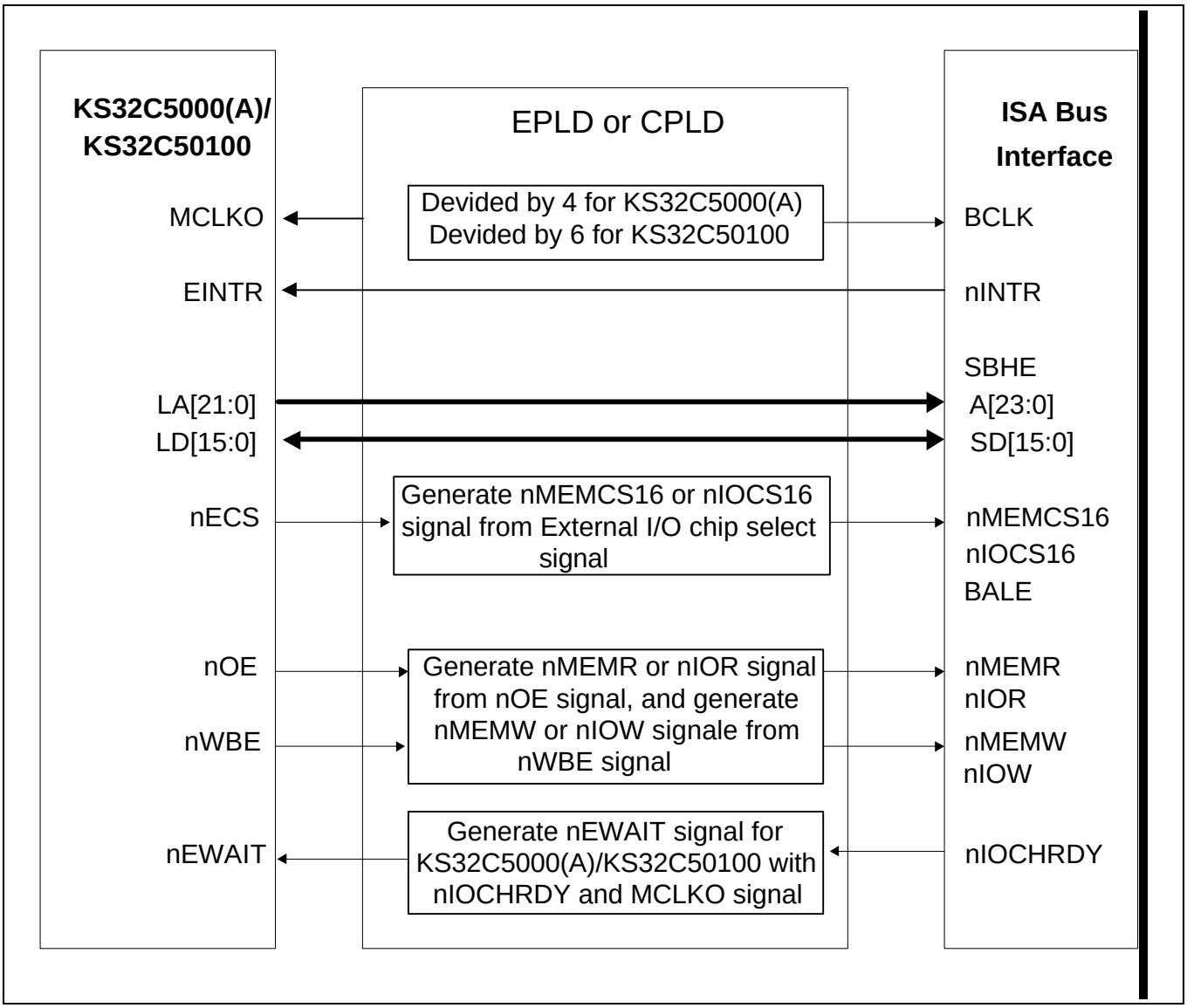


Figure 4-16. ISA Bus Interface Design

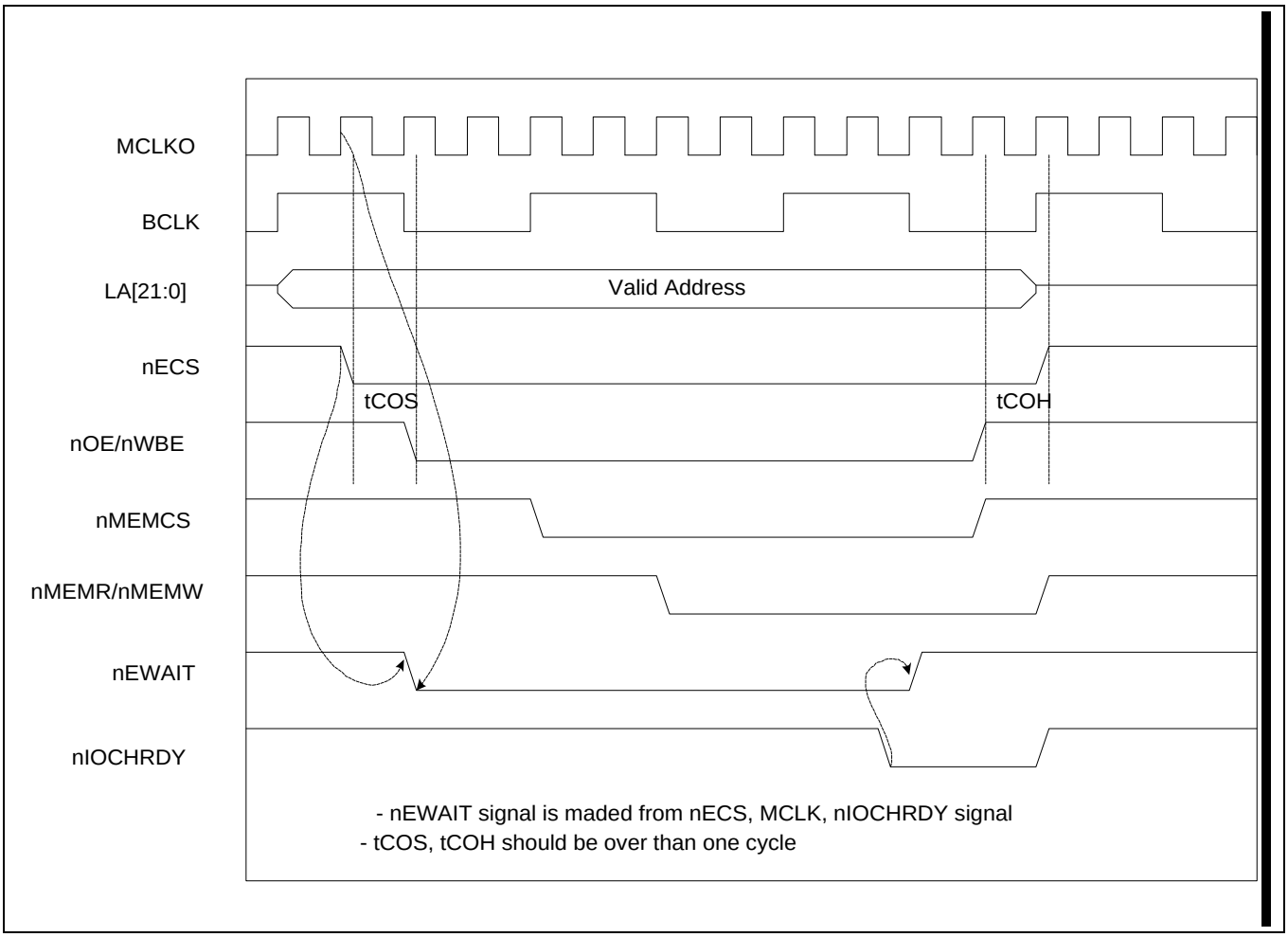


Figure 4-17. ISA Bus Interface Timing Diagram

4. Sample Design with External I/O banks and ROM/SRAM bank: PCI bus interface design

The PCI bus is 32bit synchronous bus, This block diagram is KS32C5000(A)/KS32C50100 with PCI host bridge controller,

The PCI bus general features are

- 32bit Address/Data bus
- High bandwidth (up to 132MB/sec@33MHz)
- Full multi-master capability
- Auto-configuration of devices

The PCI host bridge features are

- Generate configuration cycle
- Generate Interrupt acknowledge cycle
- Arbitration function for all PCI master

When we access PCI bus as PCI host bridge, we have three memory space,

- Memory Space
- I/O Space
- PCI Configuration Space

This three memory space make some consideration, When we design PCI host bridge interface with Samsung' s KS32C5000(A)/KS32C50100, we should be careful, Because KS32C5000(A)/KS32C50100 has total 64MB address range.

This application block diagram for PCI host bridge interface is designed with PLX9080. The Direct Master cycle is more easy to implement than Direct Slave cycle. But you should be careful when you generate address for PCI bus.

The sample block diagram is deficted in Figure 4-18.

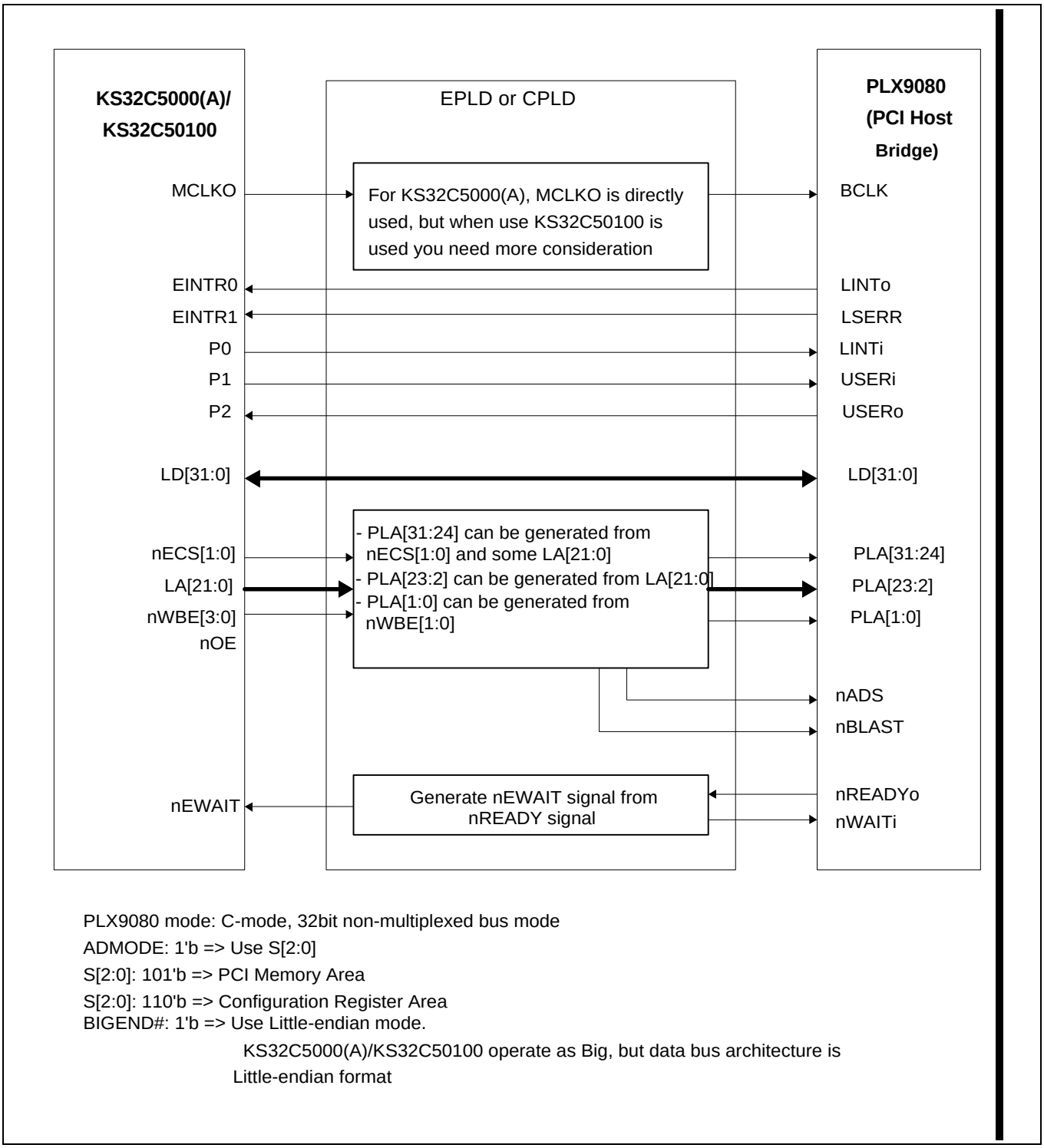


Figure 4-18. PCI bus Direct Master block diagram

The detail timing diagram for Direct master single write/read operation is described in Figure 4-19, Figure 4-20.

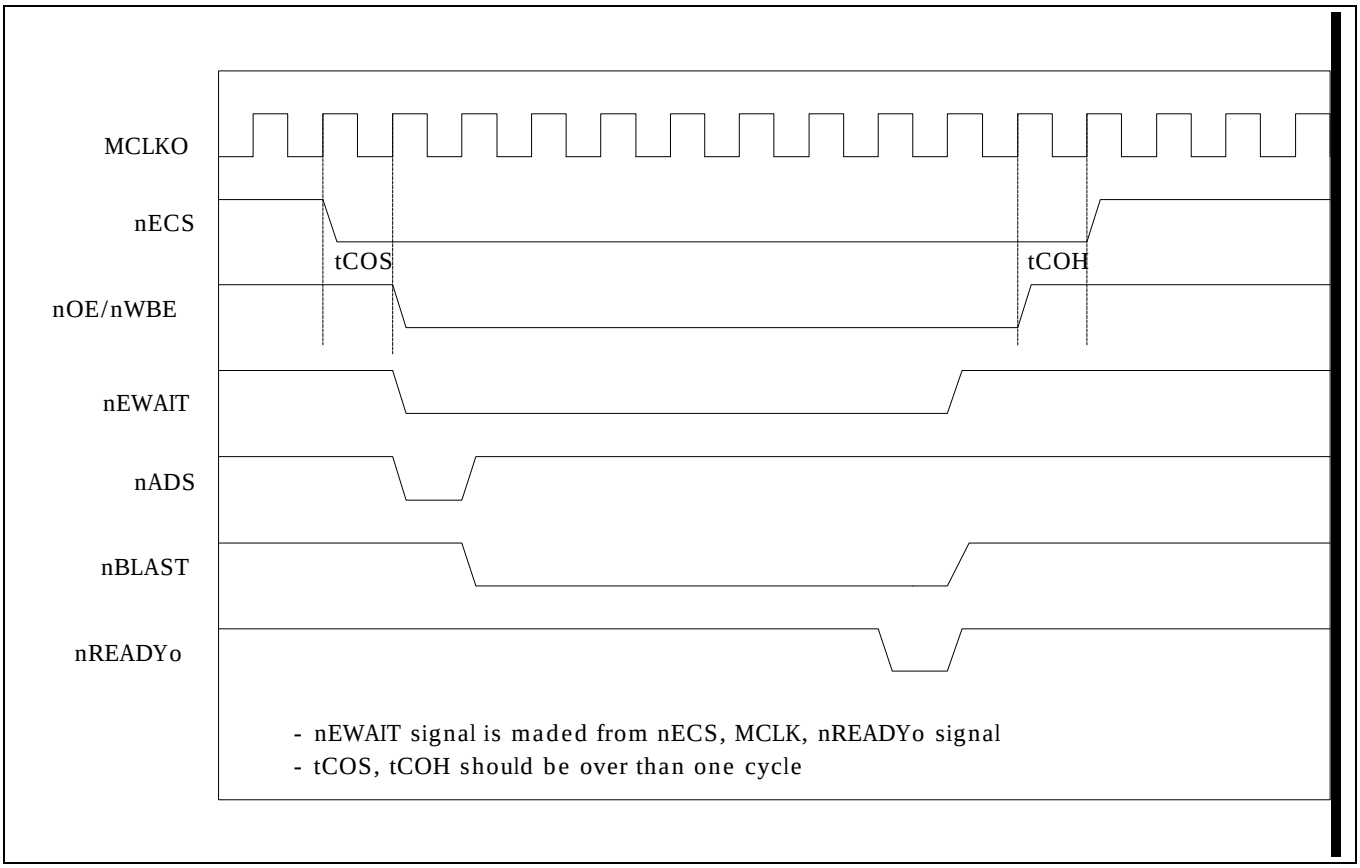


Figure 4-19. PCI bus Direct Master Write timing

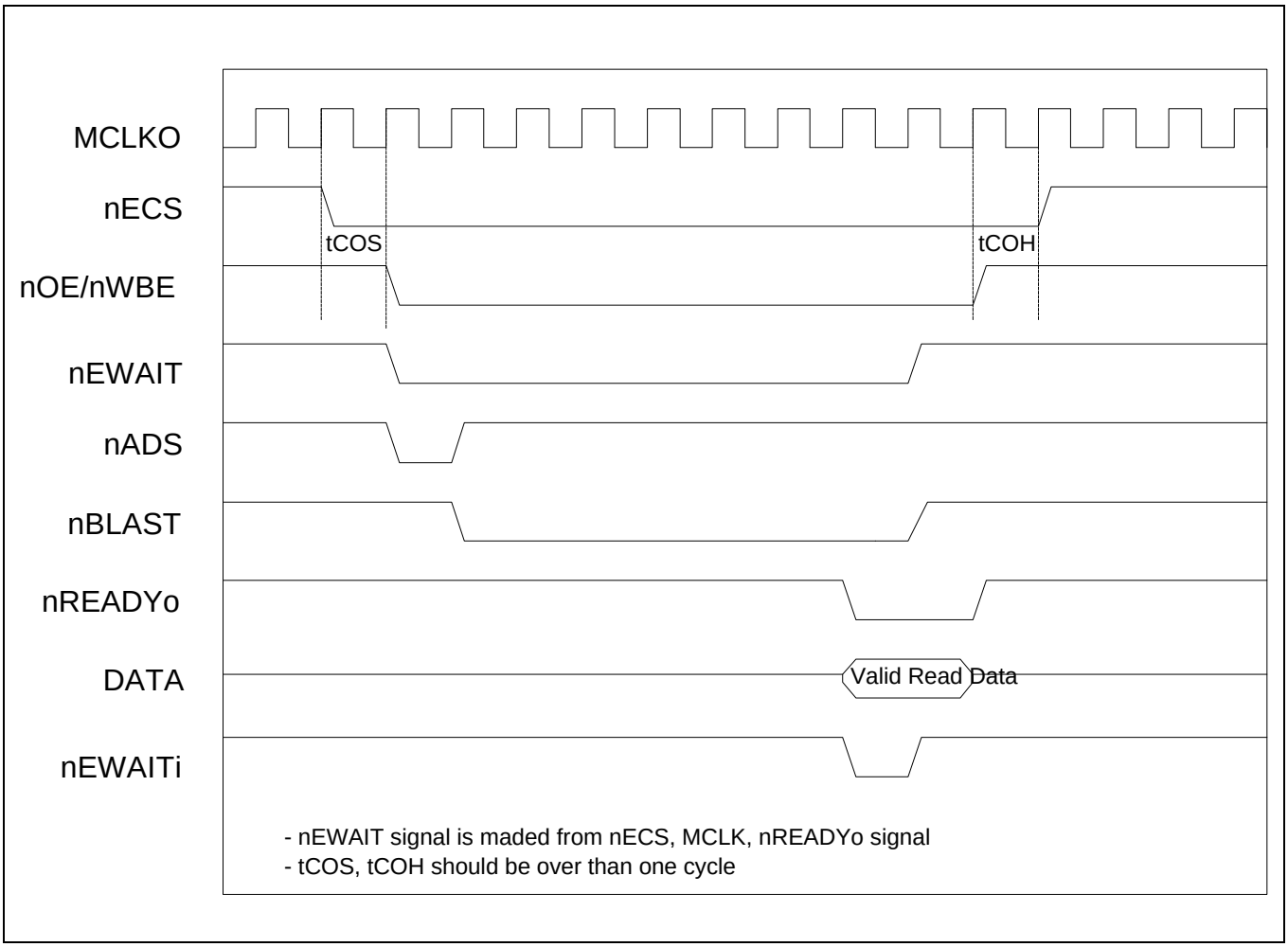


Figure 4-20. PCI bus Direct Master Read Timing

ETHERNET INTERFACE

Interface Overview

The KS32C5000(A)/KS32C50100 ethernet media access controller operates at ether 100 Mbps or 10 Mbps in halh-duplex or full-duplex mode. And the ethernet controller on the KS32C5000(A)/KS32C50100 supports both 10M/100M media independent interface (MII) and old style 10M 7-wire interface.

The KS32C5000(A)/KS32C50100 ethernet interface require physical line interface to connect a network. PHY, a physical line interface chip, is used for interface in a 10M or 100M network. PHY usually supports mediag independent interface(MII) or serial interface.

10M/100Mbps MII Interface

MII (Media Independent Interface) has control, data, and clock signal for the communication between ethernet mediag access controller (MAC) and PHY. And has a serial communication port for manage each PHY, called for Station Management.

The standard MII interface between MAC and PHY is shown in Figure 4-21.

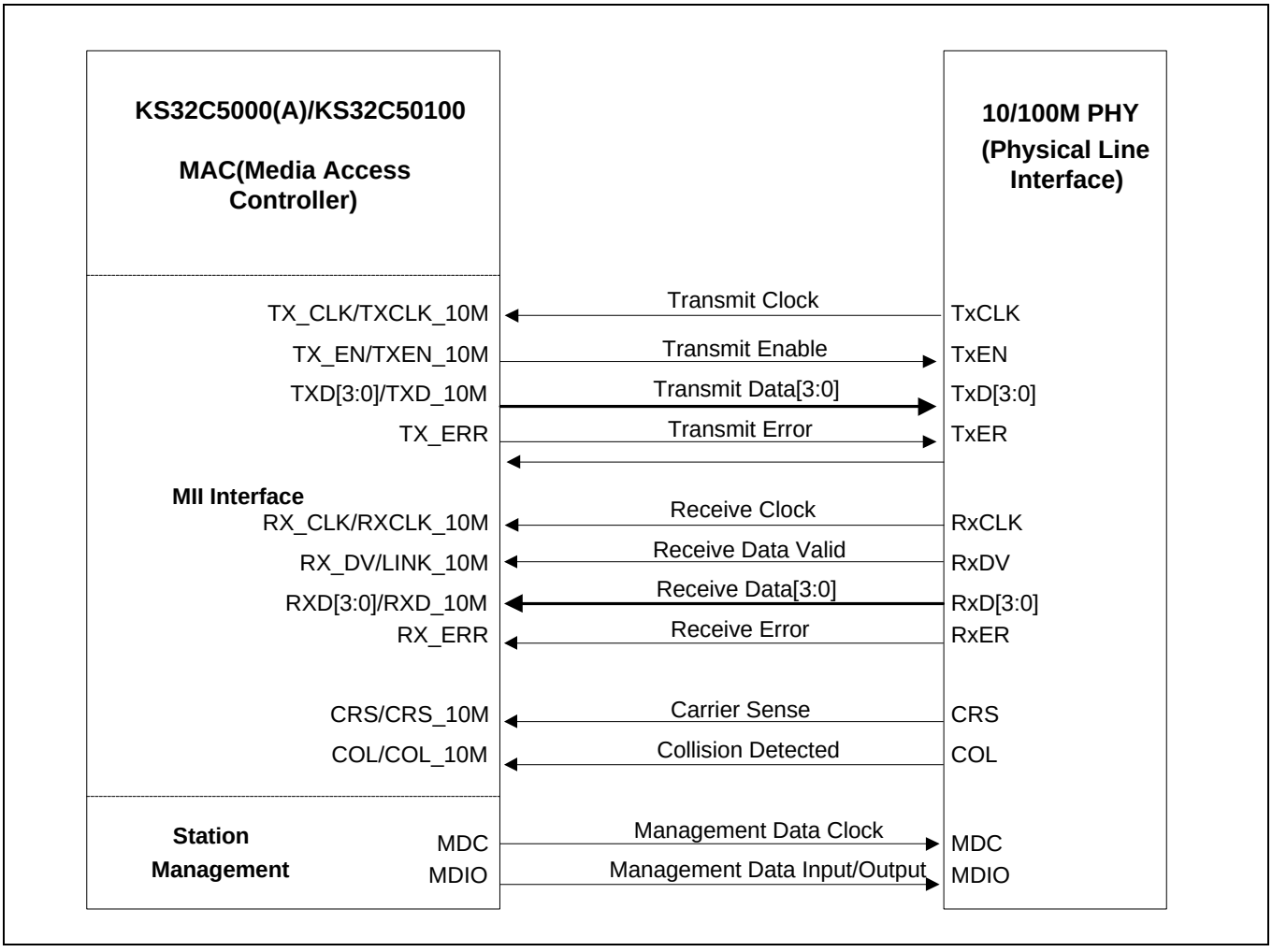


Figure 4-21. MII Connection with PHY

1. Sample Design with 10/100Mbps MII Interface PHY: ICS1890

You can refer ICS1890 datasheet for the detail specification. The configuration of ICS1890 is decided by your system usage.

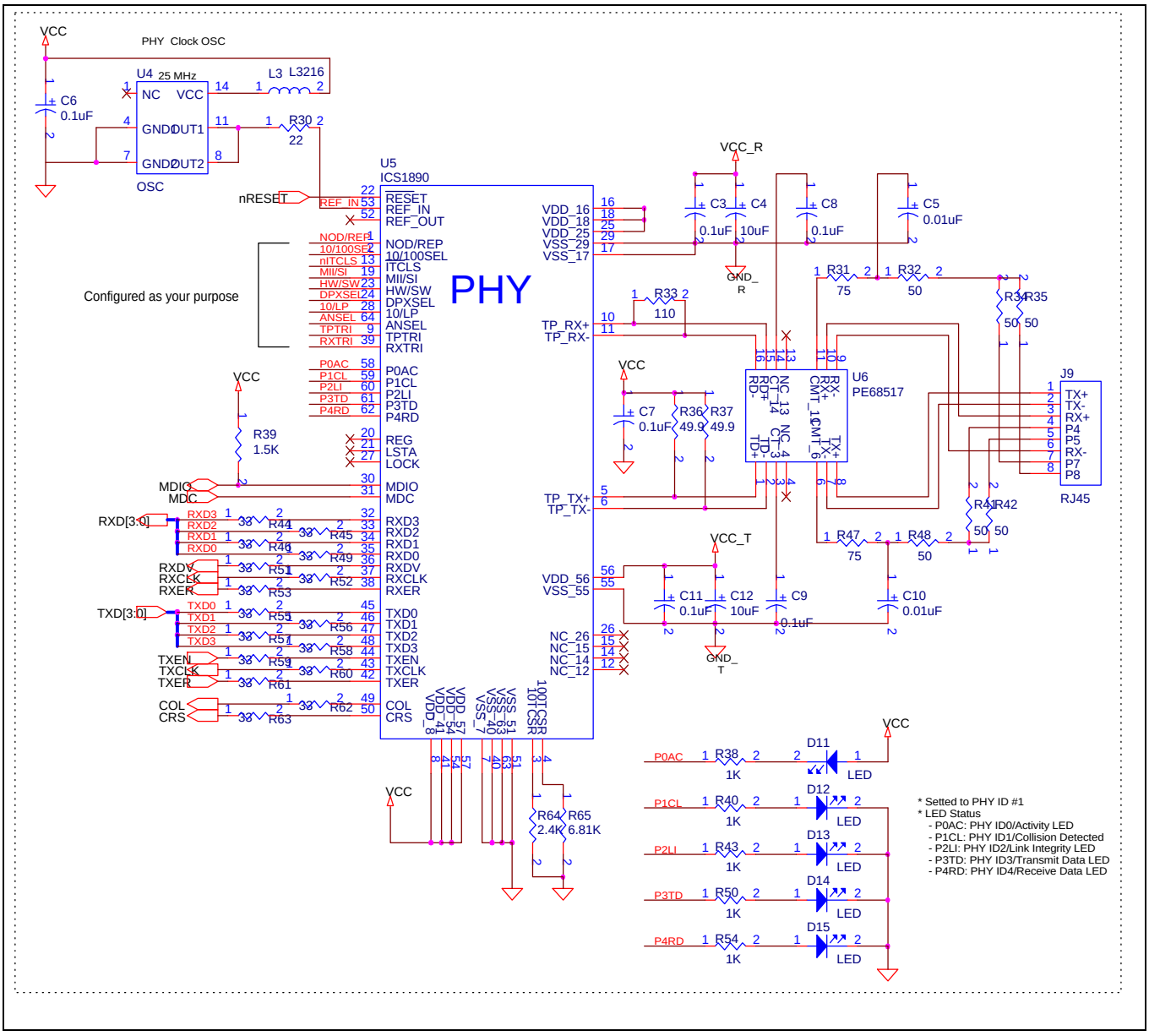


Figure 4-22. Sample Design with ICS1890 PHY

2. Sample Design with 10/100Mbps MII Interface PHY : LXT970

You can refer LXT970 datasheet for the detail specification.

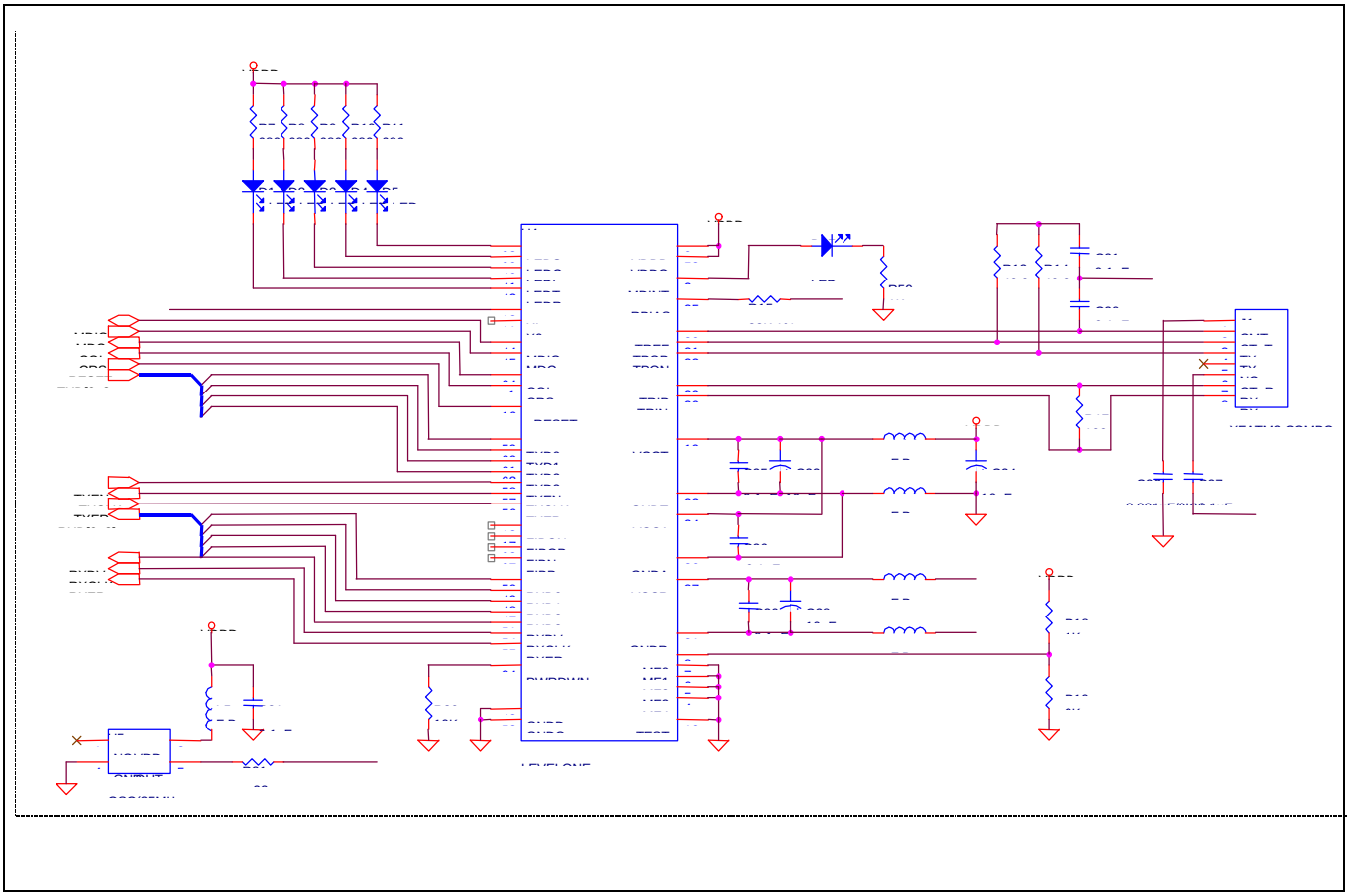


Figure 4-23. Sample Design with LXT970 PHY

3. Sample Design with 10/100Mbps MII Interface to RIC : LXT980

You can refer LXT980 datasheet for the detail specification.

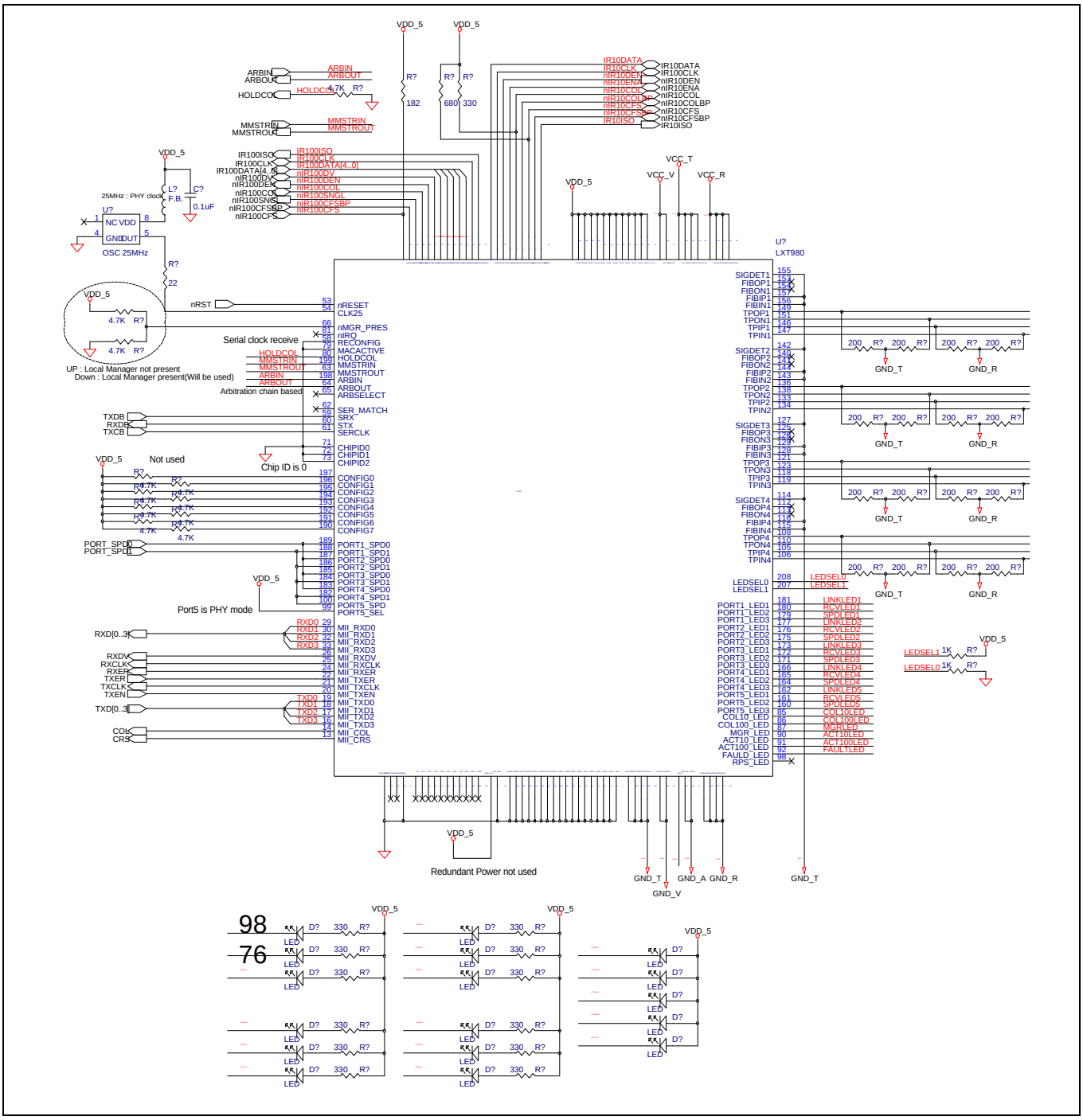


Figure 4-24. Sample Design with LXT980 4-port RIC

10M 7-wire Interface

7-wire interface only for 10M ethernet has control, data, and clock signal for the communication between ethernet mediag access controller (MAC) and PHY.

The standard 7-wire interface between MAC and PHY is shown in Figure 4-25.

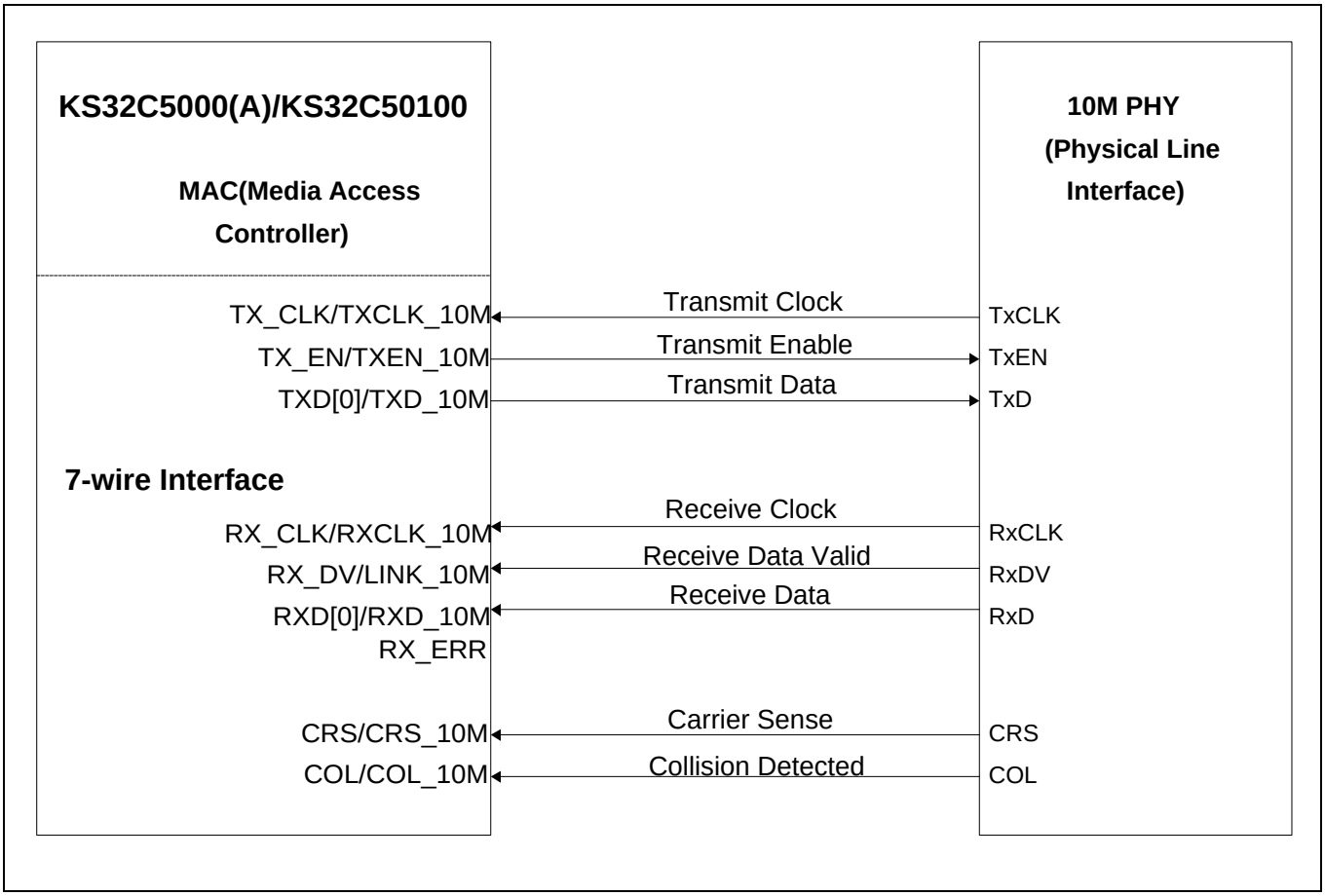


Figure 4-25. 7-Wire Interface with 10M PHY

1. Sample Design with 10Mbps 7-wire Interface: LXT901A/970A

You can refer LXT901/LXT970A datasheet for the detail specification.

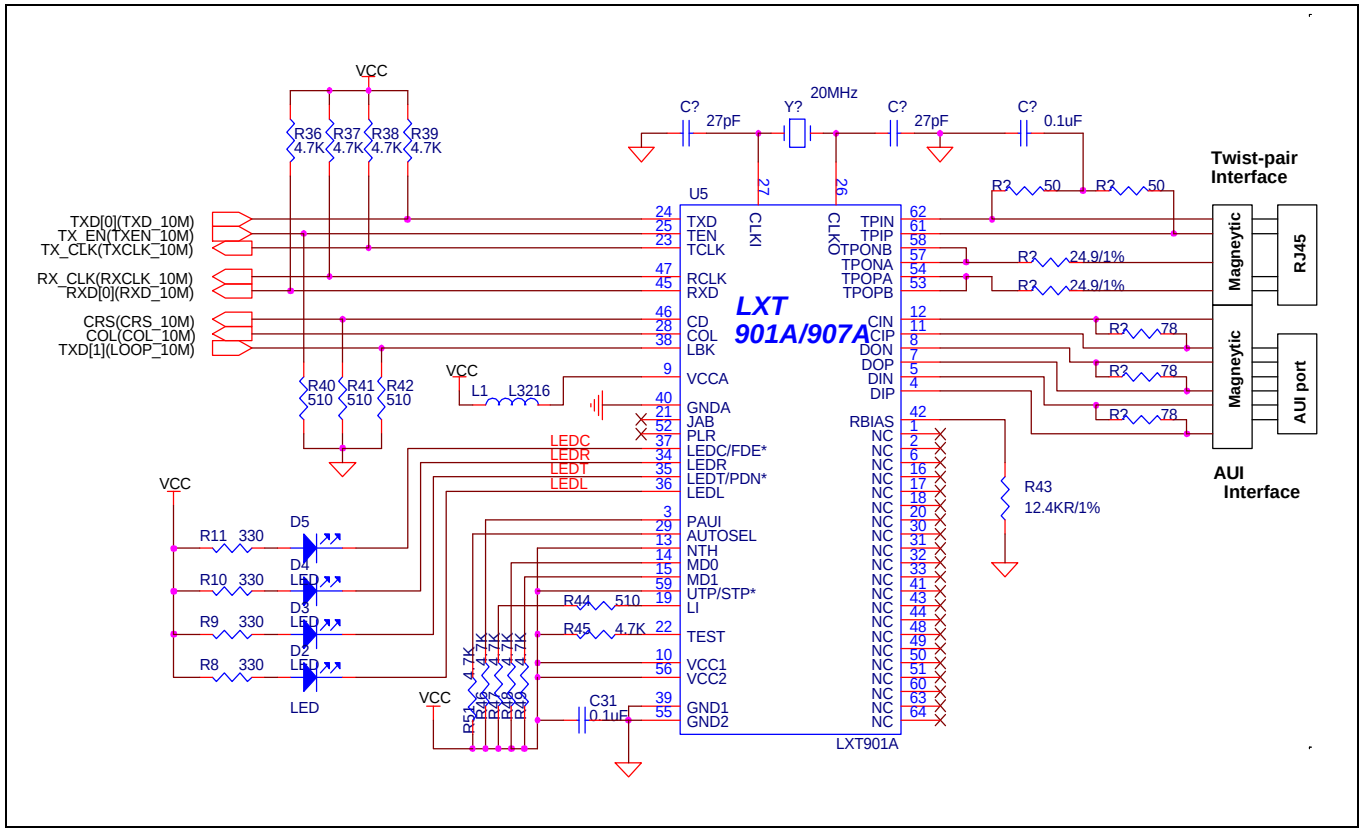


Figure 4-26. Sample Design with LXT901A/970A (Auto Port Select Support)

SYSTEM DESIGN WITH DEBUGGER SUPPORT

EmbeddedICE Macrocell and EmbeddedICE Interface

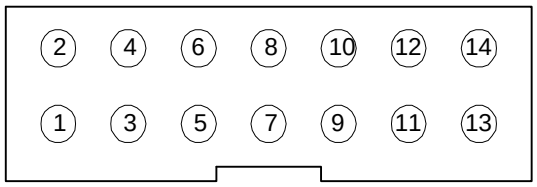
The KS32C5000(A)/KS32C50100 has an EmbeddedICE macrocell that provides debug support fro ARM cores. The EmbeddedICE macrocell is programmed in serial using the TAP(Test Access Port) controller on the KS32C5000(A)/KS32C50100. The EmbeddedICE interface is a JTAG protocol conversion unit. It translates a debug protocol message generated by the debugger into a JTAG signal which is sent to the built-in serial and parallel ports.

JTAG port for EmbeddedICE Interface

When you build a system with the KS32C5000(A)/KS32C50100 EmbeddedICE interface. You should design a JTAG port for EmbeddedICE interface. Usually, the interface connector is a 14-way box header, and this plug is connected to the EmbeddedICE interface module using 14-way IDC cable.

The JTAG port signals, nTRST,TDI,TMS, are internally pulled high, while TDO is internally pulled low which makes pull-up, and pull-down resisters unnecessary.

The pin configuration and a sample design are described in Figure 4-27, 4-28, respectively.



Pin	Name	Function
1,13	SPU	Connected to VDD through 33 or 0 ohm resister*
3	nTRST	Test reset, active low
5	TDI	Test data in
7	TMS	Test mode select
9	TCK	Test clock
11	TDO	Test data out
12	nICERST	Connected to VDD through 10K ohm resister
2,4,6,8,10,14	VSS	System Ground

* 33 ohm for EmbeddedICE, 0 ohm for Multi-ICE

Figure 4-27. EmbeddedICE Interface JTAG Connector

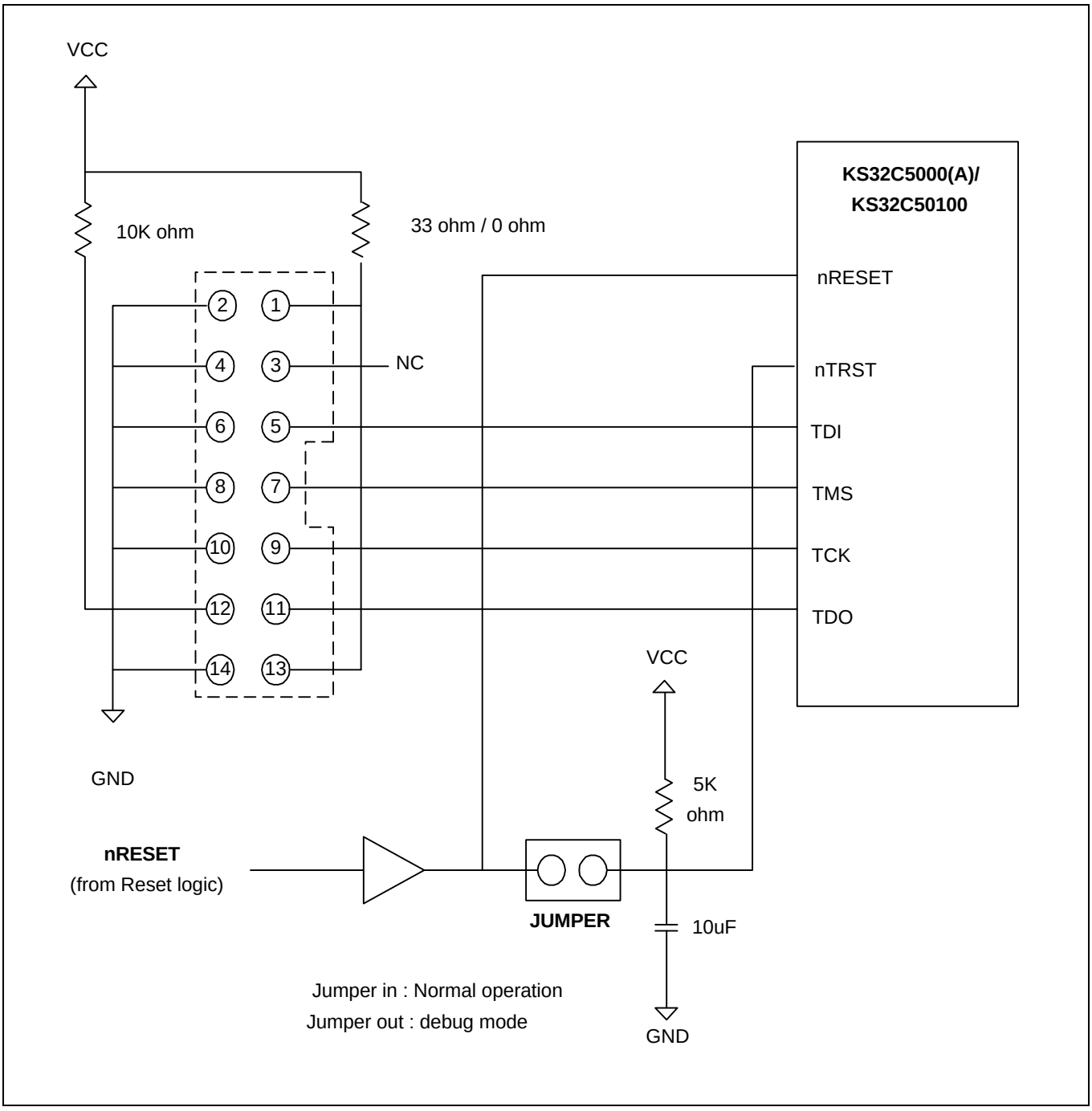


Figure 4-28. EmbeddedICE Interface Design Example

CHECK LIST FOR SYSTEM DESIGN WITH KS32C5000(A)/KS32C50100

When you design a system with the KS32C5000(A)/KS32C50100, you should check a number of items to build a good system. The check list is described below.

- The B0SIZE[1:0] signals are correctly configured to your boot ROM size.
- An nEWAIT signal has a pull-up resister.
- An ExtMREQ signal has a pull-down resister.
- Address bus interface to a memory system, that has a different internal and external address bus.
- There is a JTAG port for debugging.
- The reset logic for debugging interface. specially, nTRST and nRESET pin.
- PLL logic for KS32C50100

NOTES

5 OTHER TECHNICAL ISSUES

HOW TO RUN ROM CODE ON DRAM?

When we access ROM or Flash ROM, the access cycle is slow, so it makes system performance reduction. Specially, ARM core use base of boot ROM area, from 0x00 to 0x1C, as a exception vector area, so CPU should access the base address of boot ROM area when interrupt is issued. In our diagnostic code also use the base address of boot ROM, as an exception vector area. That is, when interrupt is occurred, CPU branch to exception area to fetch exception handler area.

Now, we' ll introduce the method to DRAM as a boot area. When use DRAM as boot area, the access cycle is reduced than ROM device access cycle, so the system performance is grows up.

THE PROCEDURE TO MAKE DRAM AS BOOT AREA

At the first time, ROM bank 0 used as boot ROM area. After boot, the all code on ROM bank 0 is copied to DRAM. The first all memory map is following the SNDS100 diagnostic code, ROM bank 0 base address is 0x00000000, and DRAM bank 0 base address is 0x1000000. After copy the code to DRAM, change the memory map to DRAM bank 0 as address 0x00000000, and ROM bank 0 as 0x1000000. The detail procedure to make DRAM as boot area is shown in Figure 5-1.

After doing this procedure, all boot code is running on the DRAM, even though exception vector table. The memory map difference between first boot and after change memory map is shown in Figure 5-2.

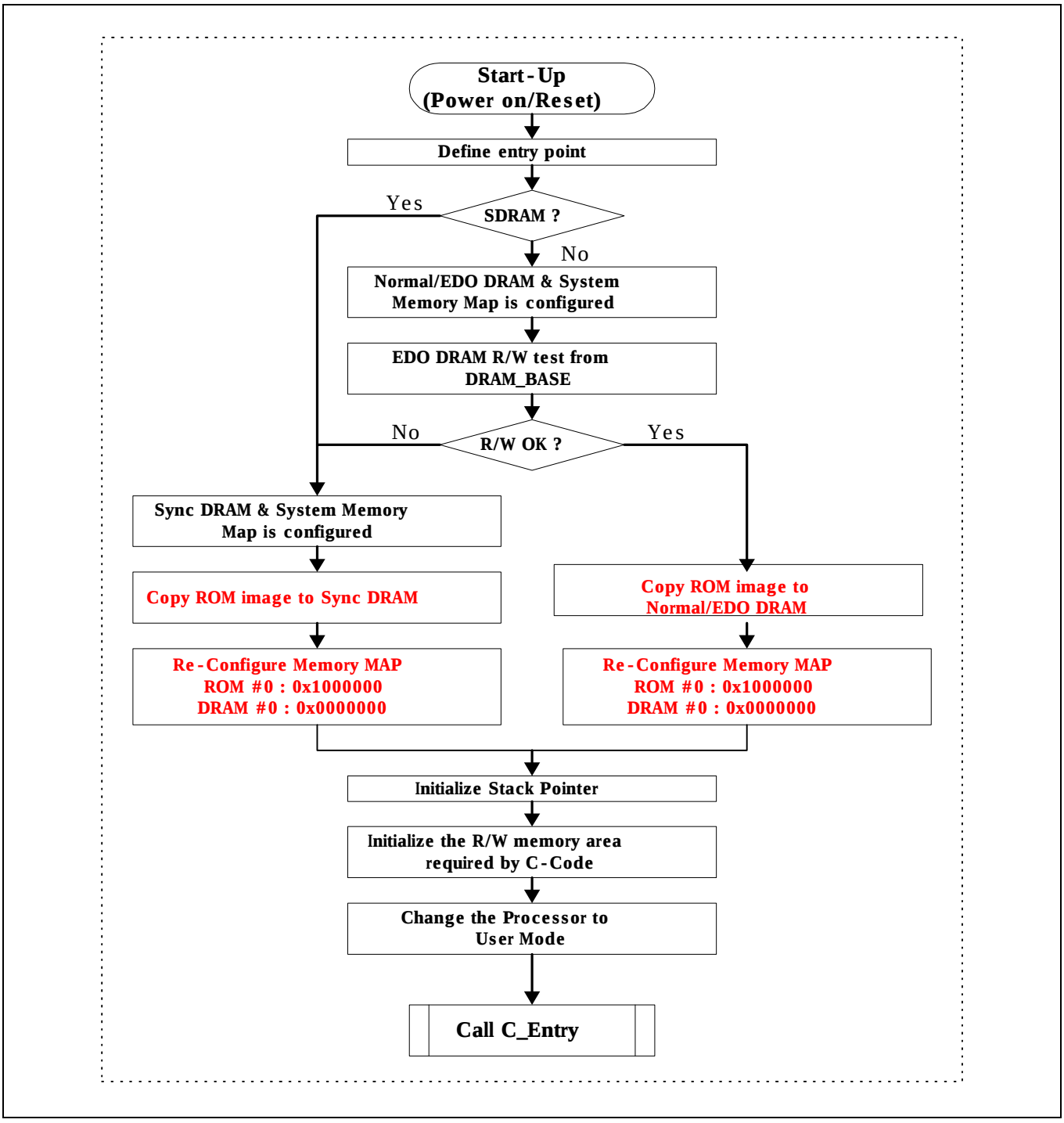


Figure 5-1. The Boot Procedure When DRAM as Boot Area

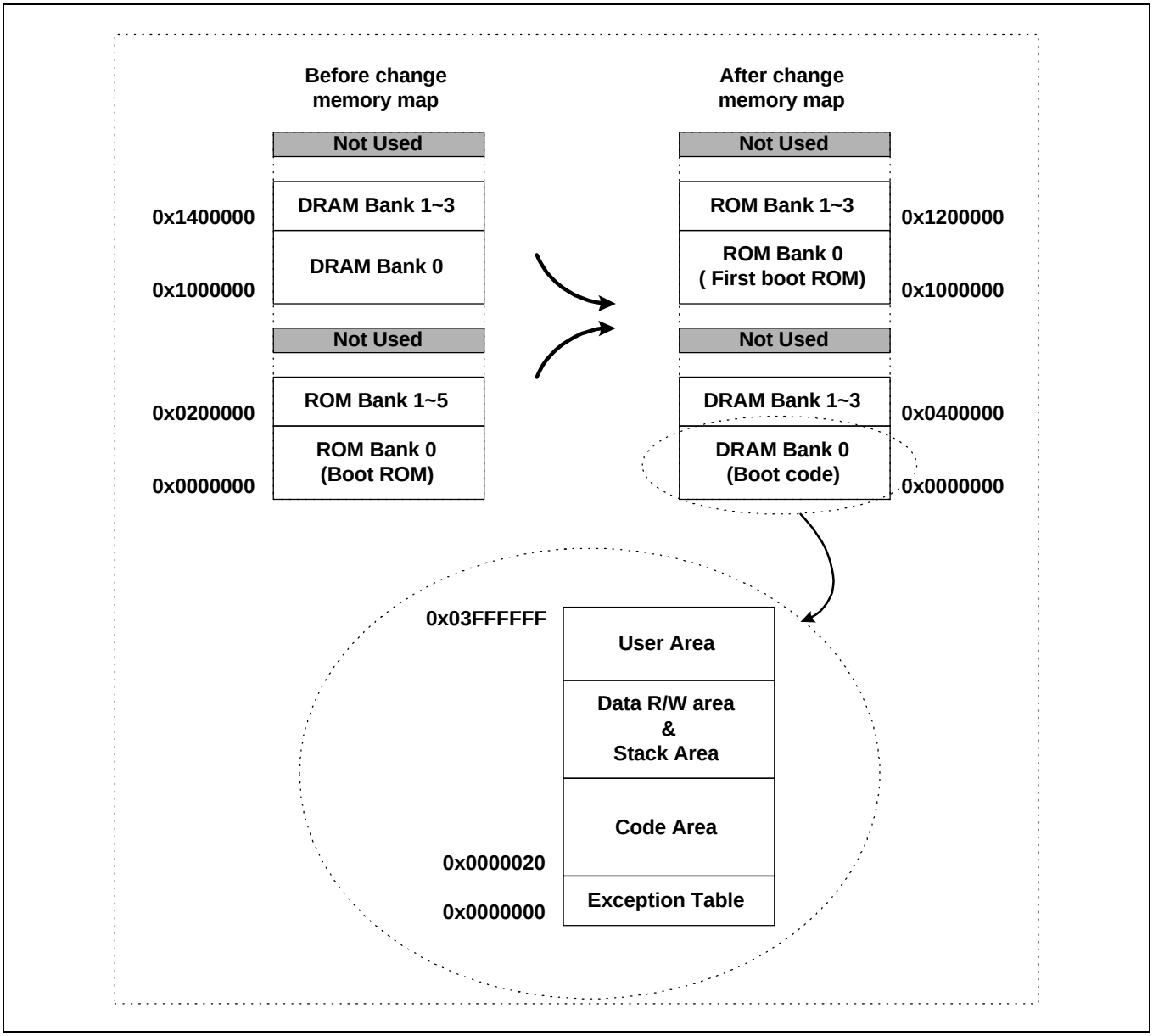


Figure 5-2. The System Memory Map When DRAM as Boot Area

THE ASSEMBLER SOURCE CODE TO MAKE DRAM AS BOOT AREA

The assembler start-up source code to make dram as boot area is some different from original SNDS100 start-up code. Some code is omitted from original SNDS100 start-up code, and some code is attached. The source code for make DRAM as boot area is described in below Listing 5-1, also in the source code, you can found the different point between original one, and the make DRAM as boot area.

Listing 5-1. Start-up Code to Make DRAM as BOOT Area (INIT.S)

```
        GET memory.a
        GET snds.a
        IMPORT  |Image$$RO$$Base|   ; Base of ROM code (=start of ROM data)
        IMPORT  |Image$$RO$$Limit|  ; End of ROM code (=start of ROM data)
        IMPORT  |Image$$RW$$Base|   ; Base of RAM to initialise
        IMPORT  |Image$$RW$$Limit|  ; Limit of RAM to initialise
        IMPORT  |Image$$ZI$$Base|   ; Base and limit of area
        IMPORT  |Image$$ZI$$Limit|  ; to zero initialise

        AREA    Init, CODE, READONLY

; --- Define entry point
      EXPORT  __main  ; defined to ensure that C runtime system
__main              ; is not linked in
      ENTRY
; --- Setup interrupt / exception vectors
   IF :DEF: ROM_AT_ADDRESS_ZERO
; If the ROM is at address 0 this is just a sequence of branches
      B       Reset_Handler
      B       SystemUndefinedHandler
      B       SystemSwiHandler
      B       SystemPrefetchHandler
      B       SystemAbortHandler
      NOP                     ; Reserved vector
      B       SystemIrqHandler
      B       SystemFiqHandler
   ELSE
; Otherwise we copy a sequence of LDR PC instructions over the vectors
; (Note: We copy LDR PC instructions because branch instructions
; could not simply be copied, the offset in the branch instruction
; would have to be modified so that it branched into ROM. Also, a
; branch instructions might not reach if the ROM is at an address
; > 32M).
      MOV     R8, #0
      ADR     R9, Vector_Init_Block
      LDMIA   R9!, {R0-R7}
      STMIA   R8!, {R0-R7}
      LDMIA   R9!, {R0-R7}
      STMIA   R8!, {R0-R7}

; Now fall into the LDR PC, Reset_Addr instruction which will continue
; execution at 'Reset_Handler'
```

```
Vector_Init_Block
      LDR    PC, Reset_Addr
      LDR    PC, Undefined_Addr
      LDR    PC, SWI_Addr
      LDR    PC, Prefetch_Addr
      LDR    PC, Abort_Addr
      NOP
      LDR    PC, IRQ_Addr
      LDR    PC, FIQ_Addr

Reset_Addr     DCD    Reset_Handler
Undefined_Addr DCD    SystemUndefinedHandler
SWI_Addr       DCD    SystemSwiHandler
Prefetch_Addr  DCD    SystemPrefetchHandler
Abort_Addr     DCD    SystemAbortHandler
               DCD    0          ; Reserved vector
IRQ_Addr       DCD    SystemIrqHandler
FIQ_Addr       DCD    SystemFiqHandler
  ENDIF

          AREA Main, CODE, READONLY

;=========================================================
; The Reset Entry Point
;=========================================================
Reset_Handler                ;/* Reset Entry Point */

          ;=====================================
          ; LED Display
          ;=====================================
          LDR      r1, =IOPMOD
          LDR      r0, =0xFF
          STR      r0, [r1]

          LDR      r1, =IOPDATA
          LDR      r0, =0x55
          STR      r0, [r1]

          ;=====================================
          ; Setup Special Register
          ;=====================================
          LDR      r0, =0x3FF0000      ; Read SYSCFG register value
          LDR      r1,[r0]             ; To idetify DRAM type
          LDR      r2, =0x80000000
          AND      r0, r1, r2      ; Mask DRAM type mode bit
          CMP      r0, r2
          BNE      EDO_DRAM_CONFIGURATION
          B        SYNC_DRAM_CONFIGURATION ; only when KS32C50100

          ;=================================================
          ; Special Register Configuration for EDO mode DRAM
          ; When KS32C5000 and KS32C50100
```

```
        ;=================================================
EDO_DRAM_CONFIGURATION
        LDR     r0, =0x3FF0000
        LDR     r1, =0x3FFFF90    ; SetValue = 0x3FFFF91
        STR     r1, [r0]          ; Cache,WB disable
                  ; Start_addr = 0x3FF00000

        ;ROM and RAM Configuration(Multiple Load and Store)
        ADRL    r0, SystemInitData
        LDMIA   r0, {r1-r12}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCntr Offset : 0x3010
        STMIA   r0, {r1-r12}

        LDR     r1,=DRAM_BASE
        STR     r1,[r1]    ; [DRAM_BASE] = DRAM_BASE
        LDR     r2,[r1]           ; Read DRAM Data
        CMP     r2,r1
        BNE     SYNC_DRAM_CONFIGURATION

        ;=============================================================
        ; Copy ROM image to EDO DRAM, Change ROM and DRAM Base pointer
        ;=============================================================
ROM2DRAM_COPY_START
        LDR     r0, =|Image$$RO$$Base|     ; Get pointer to ROM data
        LDR     r1, =|Image$$RW$$Limit|  ; and RAM copy
        LDR     r2, =DRAM_BASE             ; Copy DRAM area base
        SUB     r1, r1, r0          ; [r1] is loop count
        ADD     r1, r1, #4; [r1] is loop count

ROM2DRAM_COPY_LOOP
        LDR     r3, [r0], #4
        STR     r3, [r2], #4
        SUBS    r1, r1, #4; Down Count
        BNE     ROM2DRAM_COPY_LOOP

        ;====================================
        ; Change Base address of ROM and DRAM
        ;====================================
        ADRL    r0, SystemInitData_S
        LDMIA   r0, {r1-r12}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCntr Offset : 0x3010
        STMIA   r0, {r1-r12}

        B       INITIALIZE_STACK

        ;=================================================
        ; Special Register Configuration for SYNC DRAM
        ; Only when KS32C50100
        ;=================================================
SYNC_DRAM_CONFIGURATION
        LDR     r0, =0x3FF0000
        LDR     r1, =0x83FFFF90   ; SetValue = 0x83FFFF91
        STR     r1, [r0]          ; Cache,WB disable
```

```
                        ; Start_addr = 0x3FF00000

        ;ROM and RAM Configuration(Multiple Load and Store)
        ADRL    r0, SystemInitDataSDRAM
        LDMIA   r0, {r1-r12}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCntr Offset : 0x3010
        STMIA   r0, {r1-r12}

        ;=============================================================
        ; Copy ROM image to SYNC DRAM, Change ROM and DRAM Base pointer
        ;=============================================================
ROM2SDRAM_COPY_START
        LDR     r0, =|Image$$RO$$Base|    ; Get pointer to ROM data
        LDR     r1, =|Image$$RO$$Limit|  ; and RAM copy
        LDR     r2, =DRAM_BASE            ; Copy DRAM area base
        SUB     r1, r1, r0                ; [r1] is loop count
        ADD     r1, r1, #4              ; [r1] is loop count

ROM2SDRAM_COPY_LOOP
        LDR     r3, [r0], #4
        STR     r3, [r2], #4
        SUBS    r1, r1, #4              ; Down Count
        BNE     ROM2SDRAM_COPY_LOOP

        ;====================================
        ; Change Base address of ROM and DRAM
        ;====================================
        ADRL    r0, SystemInitDataSDRAM_S
        LDMIA   r0, {r1-r12}
        LDR     r0, =0x3FF0000 + 0x3010 ; ROMCntr Offset : 0x3010
        STMIA   r0, {r1-r12}

        ;====================================
        ; Initialise STACK
        ;====================================
INITIALIZE_STACK
        MRS     r0, cpsr
        BIC     r0, r0, #LOCKOUT | MODE_MASK
        ORR     r2, r0, #USR_MODE

        ORR     r1, r0, #LOCKOUT | FIQ_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =FIQ_STACK

        ORR     r1, r0, #LOCKOUT | IRQ_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =IRQ_STACK

        ORR     r1, r0, #LOCKOUT | ABT_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
```

```
        LDR     sp, =ABT_STACK

        ORR     r1, r0, #LOCKOUT | UDF_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =UDF_STACK

        ORR     r1, r0, #LOCKOUT | SUP_MODE
        MSR     cpsr, r1
        MSR     spsr, r2
        LDR     sp, =SUP_STACK   ; Change CPSR to SVC mode

        ;=====================================
        ; Initialise memory required by C code
        ;=====================================
        LDR  r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
        LDR  r1, =|Image$$RW$$Base|  ; and RAM copy
        LDR  r3, =|Image$$ZI$$Base|  ; Zero init base => top of initialised data
        CMP     r0, r1          ; Check that they are different
        BEQ     %1
0       CMP     r1, r3          ; Copy init data
        LDRCC   r2, [r0], #4
        STRCC   r2, [r1], #4
        BCC     %0
1       LDR     r1, =|Image$$ZI$$Limit| ; Top of zero init segment
        MOV     r2, #0
2       CMP     r3, r1          ; Zero init
        STRCC   r2, [r3], #4
        BCC     %2

        ;=================================================
        ; Now change to user mode and set up user mode stack.
        ;=================================================
    MRS     r0, cpsr
        BIC     r0, r0, #LOCKOUT | MODE_MASK
        ORR     r1, r0, #USR_MODE
        MSR     cpsr, r0
        LDR     sp, =USR_STACK

;   /* Call C_Entry application routine with a pointer to the first */
;   /* available memory address after ther compiler's global data   */
;   /* This memory may be used by the application.                  */
        ;============================
        ; Now we enter the C Program
        ;============================

    IMPORT  C_Entry
    BL      C_Entry

;=========================================
; Exception Vector Function Definition
; Consist of function Call from C-Program.
;=========================================
```

```
SystemUndefinedHandler
        IMPORT ISR_UndefHandler
        STMFD  sp!, {r0-r12}
        B      ISR_UndefHandler
        LDMFD  sp!, {r0-r12, pc}^

SystemSwiHandler
        STMFD  sp!, {r0-r12,lr}
        LDR    r0, [lr, #-4]
        BIC    r0, r0, #0xff000000
        CMP    r0, #0xff
        BEQ    MakeSVC
        LDMFD  sp!, {r0-r12, pc}^
MakeSVC
        MRS    r1, spsr
        BIC    r1, r1, #MODE_MASK
        ORR    r2, r1, #SUP_MODE
        MSR    spsr, r2
        LDMFD  sp!, {r0-r12, pc}^

SystemPrefetchHandler
        IMPORT ISR_PrefetchHandler
        STMFD  sp!, {r0-r12, lr}
        B      ISR_PrefetchHandler
        LDMFD  sp!, {r0-r12, lr}
        ;ADD   sp, sp, #4
        SUBS   pc, lr, #4

SystemAbortHandler
        IMPORT ISR_AbortHandler
        STMFD  sp!, {r0-r12, lr}
        B      ISR_AbortHandler
        LDMFD  sp!, {r0-r12, lr}
        ;ADD   sp, sp, #4
        SUBS   pc, lr, #8

SystemReserv
        SUBS   pc, lr, #4

SystemIrqHandler
        IMPORT ISR_IrqHandler
        STMFD  sp!, {r0-r12, lr}
        BL     ISR_IrqHandler
        LDMFD  sp!, {r0-r12, lr}
        SUBS   pc, lr, #4

SystemFiqHandler
        IMPORT ISR_FiqHandler
        STMFD  sp!, {r0-r7, lr}
        BL     ISR_FiqHandler
        LDMFD  sp!, {r0-r7, lr}
        SUBS   pc, lr, #4
```

```
        AREA ROMDATA, DATA, READONLY

;=====================================================
; DRAM System Initialize Data(KS32C5000 and KS32C50100)
;=====================================================
SystemInitData
        DCD rEXTDBWTH           ; DRAM1(Half), ROM5(Byte), ROM1(Half), else 32bit
        DCD rROMCON0      ; 0x0000000 ~ 0x01FFFFF, ROM0,4Mbit,2cycle
        DCD rROMCON1      ;
        DCD rROMCON2      ; 0x0400000 ~ 0x05FFFFF, ROM2
        DCD rROMCON3      ; 0x0600000 ~ 0x07FFFFF, ROM3
        DCD rROMCON4      ; 0x0800000 ~ 0x09FFFFF, ROM4
        DCD rROMCON5      ;
        DCD rDRAMCON0     ; 0x1000000 ~ 0x13FFFFF, DRAM0 4M,
        DCD rDRAMCON1           ; 0x1400000 ~ 0x17FFFFF, DRAM1 4M,
        DCD rDRAMCON2           ; 0x1800000 ~ 0x1EFFFFF, DRAM2 16M
        DCD rDRAMCON3           ; 0x1C00000 ~ 0x1FFFFFF
        DCD rREFEXTCON     ; External I/O, Refresh

SystemInitData_S
        DCD rEXTDBWTH           ; DRAM1(Half), ROM5(Byte), ROM1(Half), else 32bit
        DCD rROMCON0_S          ; 0x1000000 ~ 0x11FFFFF, ROM0,4Mbit,2cycle
        DCD rROMCON1_S          ; 0x1200000 ~ 0x13FFFFF, ROM0
        DCD rROMCON2_S          ; 0x1400000 ~ 0x15FFFFF, ROM2
        DCD rROMCON3_S          ; 0x1600000 ~ 0x17FFFFF, ROM3
        DCD rROMCON4_S          ; 0x1800000 ~ 0x19FFFFF, ROM4
        DCD rROMCON5_S          ; 0x1A00000 ~ 0x1BFFFFF, ROM4
        DCD rDRAMCON0_S    ; 0x0000000 ~ 0x03FFFFF, DRAM0
        DCD rDRAMCON1_S         ; 0x0400000 ~ 0x07FFFFF, DRAM1
        DCD rDRAMCON2_S         ; 0x0800000 ~ 0x0EFFFFF, DRAM2
        DCD rDRAMCON3_S         ; 0x0C00000 ~ 0x0FFFFFF
        DCD rREFEXTCON     ; External I/O, Refresh

;=====================================================
; SDRAM System Initialize Data         (KS32C50100 only)
;=====================================================
SystemInitDataSDRAM
        DCD rEXTDBWTH           ; DRAM1(Half), ROM5(Byte), ROM1(Half), else 32bit
        DCD rROMCON0      ; 0x0000000 ~ 0x01FFFFF, ROM0,4Mbit,2cycle
        DCD rROMCON1      ;
        DCD rROMCON2      ; 0x0400000 ~ 0x05FFFFF, ROM2
        DCD rROMCON3      ; 0x0600000 ~ 0x07FFFFF, ROM3
        DCD rROMCON4      ; 0x0800000 ~ 0x09FFFFF, ROM4
        DCD rROMCON5      ;
        DCD rSDRAMCON0     ; 0x1000000 ~ 0x13FFFFF, DRAM0 4M,
        DCD rSDRAMCON1          ; 0x1400000 ~ 0x17FFFFF, DRAM1 4M,
        DCD rSDRAMCON2          ; 0x1800000 ~ 0x1EFFFFF, DRAM2 16M
        DCD rSDRAMCON3          ; 0x1C00000 ~ 0x1FFFFFF
        DCD rSREFEXTCON    ; External I/O, Refresh

SystemInitDataSDRAM_S
        DCD rEXTDBWTH           ; DRAM1(Half), ROM5(Byte), ROM1(Half), else 32bit
```

```
        DCD rROMCON0_S          ; 0x1000000 ~ 0x11FFFFF, ROM0,4Mbit,2cycle
        DCD rROMCON1_S          ;
        DCD rROMCON2_S          ; 0x1400000 ~ 0x15FFFFF, ROM2
        DCD rROMCON3_S          ; 0x1600000 ~ 0x17FFFFF, ROM3
        DCD rROMCON4_S          ; 0x1800000 ~ 0x19FFFFF, ROM4
        DCD rROMCON5_S          ;
        DCD rSDRAMCON0_S ; 0x0000000 ~ 0x03FFFFF, DRAM0 4M,
        DCD rSDRAMCON1_S ; 0x0400000 ~ 0x07FFFFF, DRAM1 4M,
        DCD rSDRAMCON2_S ; 0x0800000 ~ 0x0EFFFFF, DRAM2 16M
        DCD rSDRAMCON3_S ; 0x0C00000 ~ 0x0FFFFFF
        DCD rSREFEXTCON  ; External I/O, Refresh

        ALIGN

;/*******************************************************/
        AREA SYS_STACK, NOINIT
;/*******************************************************/
        %     USR_STACK_SIZE
USR_STACK
        %     UDF_STACK_SIZE
UDF_STACK
        %     ABT_STACK_SIZE
ABT_STACK
        %     IRQ_STACK_SIZE
IRQ_STACK
        %     FIQ_STACK_SIZE
FIQ_STACK
        %     SUP_STACK_SIZE
SUP_STACK

;/*******************************************************/

        END
```

When use DRAM as boot area, then you also need to modify the "snds.a" file. This file define the memory controller base and end pointer, the source code of "snds.a" is listed Listing 5-2.

Listing 5-2. Memory Control Value Definition (SNDS.A)

```
;-------------------------------------------------------------
SOME CODE OMITTED FOR READIBILITY

;/* -> ROMCON0 : ROM Bank0 Control register */
;-------------------------------------------------------------
ROMBasePtr0     EQU  0x000:SHL:10    ;=0x0000000 ; normal mode
ROMEndPtr0      EQU  0x020:SHL:20    ;=0x0200000 ; normal mode
ROMBasePtr0_S   EQU  0x100:SHL:10   ;=0x1000000
ROMEndPtr0_S    EQU  0x120:SHL:20   ;=0x1200000

PMC0            EQU  0x0            ; 0x0=Normal ROM, 0x1=4Word Page
                                    ; 0x2=8Word Page, 0x3=16Word Page
rTpa0           EQU  (0x0:SHL:2)    ; 0x0=5Cycle, 0x1=2Cycle
```

```
                                    ; 0x2=3Cycle, 0x3=4Cycle
rTacc0      EQU  (0x6:SHL:4)    ; 0x0=Disable, 0x1=2Cycle
                                    ; 0x2=3Cycle, 0x3=4Cycle
                                    ; 0x4=5Cycle, 0x5=6Cycle
                                    ; 0x6=7Cycle, 0x7=Reserved

rROMCON0    EQU  ROMEndPtr0+ROMBasePtr0+rTacc0+rTpa0+PMC0
rROMCON0_S          EQU  ROMEndPtr0_S+ROMBasePtr0_S+rTacc0+rTpa0+PMC0
;---------------------------------------------------------------


SOME CODE OMITTED FOR READIBILITY

;/* -> DRAMCON0 : RAM Bank0 control register */
;---------------------------------------------------------------
EDO_Mode0       EQU  1              ;(EDO)0=Normal, 1=EDO DRAM
CasPrechargeTime0  EQU  1              ;(Tcp)0=1cycle,1=2cycle
CasStrobeTime0    EQU  3              ;(Tcs)0=1cycle ~ 3=4cycle
DRAMCON0Reserved   EQU  1              ; Must be set to 1

RAS2CASDelay0     EQU  1              ;(Trc)0=1cycle,1=2cycle
RASPrechargeTime0  EQU  3              ;(Trp)0=1cycle ~ 3=4clcyle
DRAMBasePtr0      EQU  0x100:SHL:10       ;=0x1000000
DRAMEndPtr0       EQU  0x140:SHL:20       ;=0x1400000
DRAMBasePtr0_S    EQU  0x000:SHL:10       ;=0x0000000
DRAMEndPtr0_S     EQU  0x040:SHL:20       ;=0x0400000
NoColumnAddr0     EQU  2              ;0=8bit,1=9bit,2=10bit,3=11bits
;---------------------------------------------------------------
Tcs0          EQU  CasStrobeTime0:SHL:1
Tcp0          EQU  CasPrechargeTime0:SHL:3
dumy0         EQU  DRAMCON0Reserved:SHL:4 ; dummy cycle

Trc0          EQU  RAS2CASDelay0:SHL:7
Trp0          EQU  RASPrechargeTime0:SHL:8
CAN0          EQU  NoColumnAddr0:SHL:30

rDRAMCON0   EQU  CAN0+DRAMEndPtr0+DRAMBasePtr0+Trp0+Trc0+Tcp0+Tcs0+dumy0+EDO_Mode0
rDRAMCON0_S         EQU
CAN0+DRAMEndPtr0_S+DRAMBasePtr0_S+Trp0+Trc0+Tcp0+Tcs0+dumy0+EDO_Mode0
;---------------------------------------------------------------------------


SRAS2CASDelay0    EQU  1              ;(Trc)0=1cycle,1=2cycle
SRASPrechargeTime0 EQU  3              ;(Trp)0=1cycle ~ 3=4clcyle
SNoColumnAddr0    EQU  0              ;0=8bit,1=9bit,2=10bit,3=11bits
SCAN0         EQU  SNoColumnAddr0:SHL:30
STrc0         EQU  SRAS2CASDelay0:SHL:7
STrp0         EQU  SRASPrechargeTime0:SHL:8
;
rSDRAMCON0    EQU   SCAN0+DRAMEndPtr0+DRAMBasePtr0+STrp0+STrc0
rSDRAMCON0_S      EQU SCAN0+DRAMEndPtr0_S+DRAMBasePtr0_S+STrp0+STrc0
;---------------------------------------------------------------
```

EXCEPTION HANDLING WHEN USE DRAM AS BOOT AREA

When use DRAM as boot area, the exception handle routine is easy to implement. At the original diagnostic code use DRAM base pointer as exception handler area, but when you use DRAM as boot area, you only need the exception vector area to going into interrupt service routine. The exception handle routine is depicted in Figure 5-3. This figure show the exception handling routine example with IRQ exception.

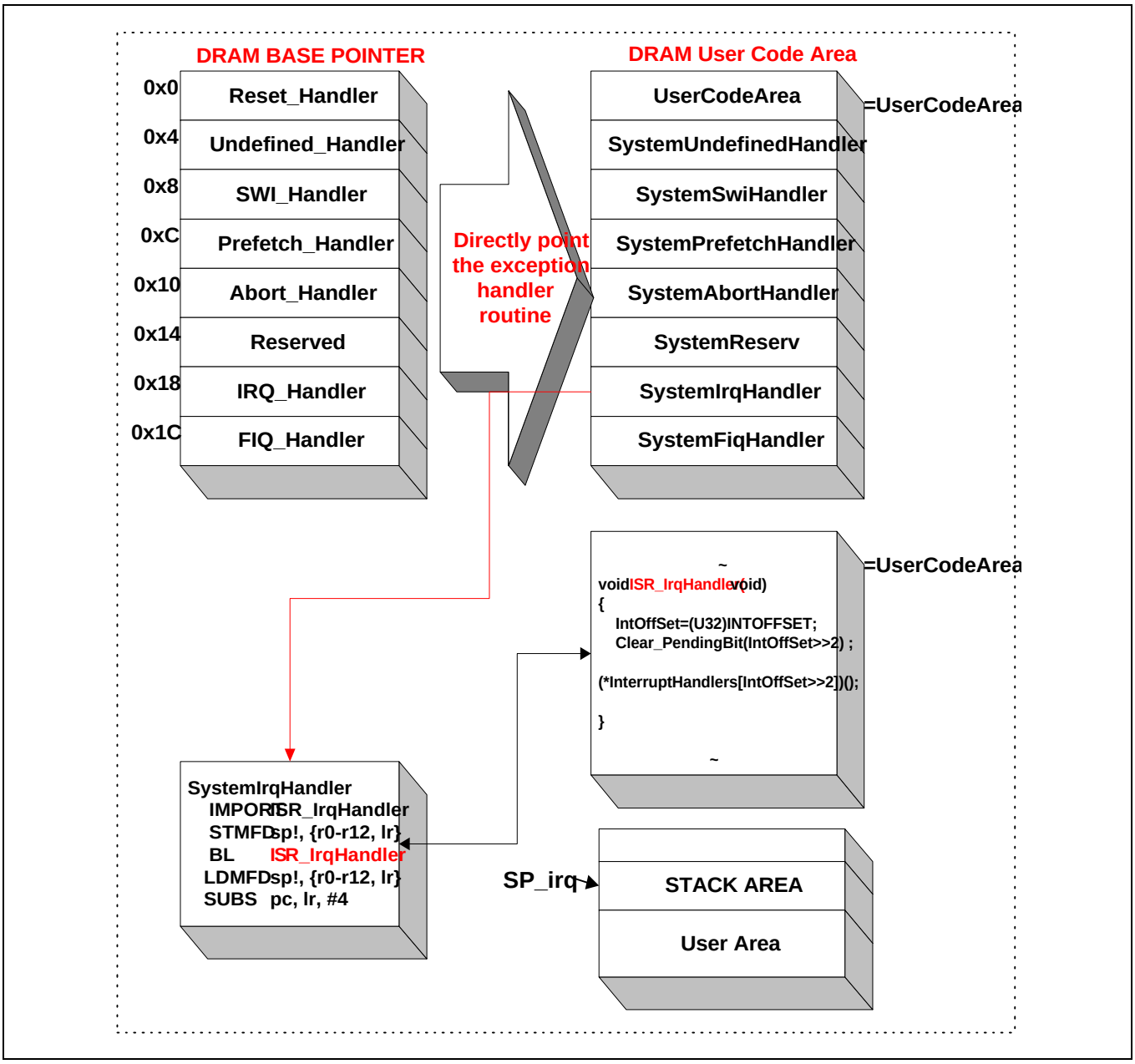


Figure 5-3. Exception Handling Process When Use DRAM as Boot Area

HOW TO DEBUG SNDS100 USING EMULATOR WITHOUT BOOT ROM?

You can evaluate SNDS100 board using circuit emulator(ex, EmbeddedICE) without boot rom. But, to do this, you have to consider two points. One is the side of hardware consideration and the other is software consideration.

That is due to the reset circuit(nRESET, nTRST) problem of NetMCU device. SNDS100 reset circuit have to be revised which is refered to "Section 4. System Design : Figure 4-28 EmbededICE interface design example".

This is came from the without boot ROM. After the power on reset or key reset, ROM bank0 will be located at address zero by default reset system memory map but there is no boot ROM. So, you have to re-map the system default memory map before downloading the application. This memory map can be changed by call initial map file at ADW' s command window.(Arm Debuger window).

HARDWARE CONSIDERATION

It is the main problem of invoking ADW (ARM Debugger) after resetting SNDS100 board that SWI interrupts the downloading of a debugging program . We find out that nTRST(62-th pin : pull-up internally) is needed to be floated. Just cutting the pattern connected to nTRST pin in SNDS100 board , you can invoke ADW and download a program without any problem and without considering time interval. (See, Figure 4-28).

SOFTWARE CONSIDERATION

After modifying SNDS100 board , you can go to the next step as following flow.

— Making a debugging image file
— Invoking ADW
— Setting the special registers of KS32C5000 series and ADW for debugging.
— Downloading Image file.
— Step into Debugging mode

The downloaded program will start and you can debug your program with the ADW.

Making a debugging image file

You need to modify files concerning memory configuration and interrupt handling.

These are init.s, memory.a and makefile

Modifying memory.a and snds.a

Some value related to memory configuration should be changed in a pre-distributed memory. a file.

— DRAM_Base symbol value

— (This means downloading start area, it is distinguished form booting code we tried)

```
DRAM_Base  EQU 0x1000000          (old)
DRAM_Base  EQU 0x300000           (new)
```

— System User Area

```
^          DRAM_Base+ExceptionSize ;
; ^        0x1000050                (old )
; ^        0x300050 ;               (new)
```

Only one symbol is needed to be changed for snds.a

```
DRAM_BASE  EQU 0x1000000          (old)
DRAM_BASE  EQU 0x300000           (new)
```

Modifying init.s

Inis.s file is simplified only for downloading DRAM. The main difference from pre-distributed one is that Memory Initialized (SystemInitData symbol) part is removed. Part 2.3 Setting Special Register for KS32C50100 will explain how to replace this role . And we overwrite IRQ_handler address at 0x18 address and FIQ_handler address at 0x1c address for interrupt mapping . This may corrupt a debugging information , but it will is correspond to only debug starting information. So you can keep debugging with ADW without regarding this.

Listing 5-3. Modifying Init.s

```
  GET memory.a
  GET snds.a

  AREA    Init, CODE, READONLY

; --- Define entry point
        EXPORT  __main  ; defined to ensure that C runtime system
__main                  ; is not linked in
        ENTRY
            B       Reset_Handler
        B       Undefined_Handler
        B       SWI_Handler
        B       Prefetch_Handler
        B       Abort_Handler
        NOP                         ; Reserved vector
        B       IRQ_Handler
        B       FIQ_Handler
```

```
;=========================================================
; The Default Exception Handler Vector Entry Pointer Setup
;=========================================================
  nop
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}


  AREA Main, CODE, READONLY

;=====================================
; The Reset Entry Point
;=====================================
Reset_Handler
```

```
;====================================
; Initialise STACK
;====================================
INITIALIZE_STACK
  MRS     r0, cpsr
  BIC     r0, r0, #LOCKOUT | MODE_MASK
  ORR     r2, r0, #USR_MODE

  ORR     r1, r0, #LOCKOUT | FIQ_MODE
  MSR     cpsr, r1
  MSR     spsr, r2
  LDR     sp, =FIQ_STACK

  ORR     r1, r0, #LOCKOUT | IRQ_MODE
  MSR     cpsr, r1
  MSR     spsr, r2
  LDR     sp, =IRQ_STACK

  ORR     r1, r0, #LOCKOUT | ABT_MODE
  MSR     cpsr, r1
  MSR     spsr, r2
  LDR     sp, =ABT_STACK

  ORR     r1, r0, #LOCKOUT | UDF_MODE
  MSR     cpsr, r1
  MSR     spsr, r2
  LDR     sp, =UDF_STACK

  ORR     r1, r0, #LOCKOUT | SUP_MODE
  MSR     cpsr, r1
  MSR     spsr, r2
  LDR     sp, =SUP_STACK   ; Change CPSR to SVC mode


;====================================
; LED Display
;====================================
  LDR     r1, =IOPMOD
  LDR     r0, =0xFF
  STR     r0, [r1]

  LDR     r1, =IOPDATA
  LDR     r0, =0x55
  STR     r0, [r1]

;====================================
; UART Setup for Console
; 38400, data= 8 bits , stop = 1 bits
;====================================
  LDR     r1, =UARTLCON0
  LDR     r0, =0x43
  STR     r0,[r1]

  LDR     r1, =UARTCONT0
  LDR     r0, =0x9
  STR     r0,[r1]

  LDR     r1, =UARTBRD0
; LDR     r0, =0x280 ; 38400 bps for 50MHz
; LDR     r0, =0x350 ; 38400 bps for 33MHz
  LDR     r0, =0x2f0 ; 38400 bps for 29.4912MHz
; ;LDR     r0, =0x160 ; 38400 bps for 14.31818MHz
  STR     r0,[r1]
```

```
  ; Print out   Boot Up Banner  to Console
  LDR       r0, =BootUpBanner0
  BL        PrintString

;  Main Memory Test (POST)
  LDR       r0, =DramPostStart
  BL        PrintString

;=================================================
; Special Register Configuration for EDO mode DRAM
;=================================================
  B         EXCEPTION_VECTOR_TABLE_SETUP

;=================================================
; Special Register Configuration for SYNC DRAM
;=================================================

;==============================
; Exception Vector Table Setup
;==============================
EXCEPTION_VECTOR_TABLE_SETUP
  LDR       r0, =HandleReset ; Exception Vector Table Memory Loc.
  LDR       r1, =ExceptionHandlerTable ; Exception Handler Assign
  MOV       r2, #8                     ; Number of Exception is 8
ExceptLoop
  LDR       r3, [r1], #4
  STR       r3, [r0], #4
  SUBS      r2, r2, #1                 ; Down Count
  BNE       ExceptLoop

;====================================
; Initialise memory required by C code
;====================================
            IMPORT  |Image$$RO$$Limit|  ; End of ROM code (=start of ROM data)
            IMPORT  |Image$$RW$$Base|   ; Base of RAM to initialise
            IMPORT  |Image$$ZI$$Base|   ; Base and limit of area
            IMPORT  |Image$$ZI$$Limit|  ; to zero initialise

            LDR  r0, =|Image$$RO$$Limit| ; Get pointer to ROM data
            LDR  r1, =|Image$$RW$$Base|  ; and RAM copy
            LDR  r3, =|Image$$ZI$$Base|  ; Zero init base => top of initialised data
            CMP     r0, r1              ; Check that they are different
            BEQ     %1
0           CMP     r1, r3              ; Copy init data
            LDRCC   r2, [r0], #4
            STRCC   r2, [r1], #4
            BCC     %0
1           LDR     r1, =|Image$$ZI$$Limit| ; Top of zero init segment
            MOV     r2, #0
2           CMP     r3, r1              ; Zero init
            STRCC   r2, [r3], #4
            BCC     %2

;=================================================
; Now change to user mode and set up user mode stack.
;=================================================
    MRS     r0, cpsr
  BIC       r0, r0, #LOCKOUT | MODE_MASK
  ORR       r1, r0, #USR_MODE
  MSR       cpsr, r0
  LDR       sp, =USR_STACK

;       /* Call C_Entry application routine with a pointer to the first */
```

```
;       /* available memory address after ther compiler's global data   */
;       /* This memory may be used by the application.                  */
;===========================
; Now we enter the C Program
;===========================

  LDR       r0, =0x98
  LDR       r1, =0x18
  LDR       r2, [r0]
  STR       r2, [r1]

;  This is for IRQ_handler

  LDR       r0, =0x9c
  LDR       r1, =0x1c
  LDR       r2, [r0]
  STR       r2, [r1]

;  This is for FIQ_handler

   IMPORT  C_Entry
   BL      C_Entry

;=========================================
; Exception Vector Function Definition
; Consist of function Call from C-Program.
;=========================================
SystemUndefinedHandler
  IMPORT ISR_UndefHandler
  STMFD  sp!, {r0-r12}
  B          ISR_UndefHandler
  LDMFD  sp!, {r0-r12, pc}^

SystemSwiHandler
  STMFD  sp!, {r0-r12,lr}
  LDR       r0, [lr, #-4]
  BIC       r0, r0, #0xff000000
  CMP       r0, #0xff
  BEQ       MakeSVC
  LDMFD  sp!, {r0-r12, pc}^
MakeSVC
  MRS       r1, spsr
  BIC       r1, r1, #MODE_MASK
  ORR       r2, r1, #SUP_MODE
  MSR       spsr, r2
  LDMFD  sp!, {r0-r12, pc}^

SystemPrefetchHandler
  IMPORT ISR_PrefetchHandler
  STMFD  sp!, {r0-r12, lr}
  B          ISR_PrefetchHandler
  LDMFD  sp!, {r0-r12, lr}
  ;ADD     sp, sp, #4
  SUBS    pc, lr, #4

SystemAbortHandler
  IMPORT ISR_AbortHandler
  STMFD  sp!, {r0-r12, lr}
  B          ISR_AbortHandler
  LDMFD  sp!, {r0-r12, lr}
  ;ADD     sp, sp, #4
  SUBS    pc, lr, #8

SystemReserv
```

```
  SUBS    pc, lr, #4

SystemIrqHandler
  IMPORT ISR_IrqHandler
  STMFD  sp!, {r0-r12, lr}
  BL     ISR_IrqHandler
  LDMFD  sp!, {r0-r12, lr}
  SUBS   pc, lr, #4

SystemFiqHandler
  IMPORT ISR_FiqHandler
  STMFD  sp!, {r0-r7, lr}
  BL     ISR_FiqHandler
  LDMFD  sp!, {r0-r7, lr}
  SUBS   pc, lr, #4

;=======================================================
;                    Utility Section
;             Print a zero terminated string
;=======================================================
PrintString
  MOV           r4, lr
  MOV           r5, r0
01      LDRB          r1, [r5],#1          ; read Byte( one character)
  AND   r0,r1, #&ff
  TST           r0, #&ff
  MOVEQ         pc, r4              ; if 0, return.
  BL            PutByte
  B             %B01

PutByte
PutByteLoop
  LDR   r3,=UARTSTAT0
  LDR   r2,[r3]
  TST   r2,#&40               ; UART_STAT_XMIT_EMPTY ?
  BEQ   PutByteLoop
  LDR   r3,=UARTTXH0
  STR   r0,[r3]               ; write to THR
  MOV   pc,lr


  AREA ROMDATA, DATA, READONLY

;=========================================
; Exception Handler Vector Table Entry Point
;=========================================
ExceptionHandlerTable
  DCD   UserCodeArea
  DCD   SystemUndefinedHandler
  DCD   SystemSwiHandler
  DCD   SystemPrefetchHandler
  DCD   SystemAbortHandler
  DCD   SystemReserv
  DCD   SystemIrqHandler
  DCD   SystemFiqHandler

  ALIGN

BootUpBanner0   DCB     &a,&d,&a,&d,"$$$   SNDS100 Boot-Up Dialog !!!",0
DramPostStart   DCB     &a,&d,"$$$   SNDS100 DRAM Post => ",0
DramPostFinish  DCB     "MB Installed !",0
CacheEnableBanner       DCB     &a,&d," $$$   8K Cache is Enabled !",0

;/*****************************************************/
```

```
  AREA SYS_STACK, NOINIT
;/*****************************************************/
        %       USR_STACK_SIZE
USR_STACK
        %       UDF_STACK_SIZE
UDF_STACK
        %       ABT_STACK_SIZE
ABT_STACK
        %       IRQ_STACK_SIZE
IRQ_STACK
        %       FIQ_STACK_SIZE
FIQ_STACK
        %       SUP_STACK_SIZE
SUP_STACK

;/*****************************************************/

  END
```

Modifying makefile

You should map Code area to address 0x0 and R/W area to a reasonable address below DRAM_Base. The following list shows the way to modify a makefile .

Listing 5-4. Makefile

```
TOOLPath        = C:\arm211a\bin
LINK    = $(TOOLPath)\armlink
ASM     = $(TOOLPath)\armasm
CC      = $(TOOLPath)\armcc
LIB     = C:\arm211a\lib\armlib.32b

ASMFLAG1        = -bi -g -apcs 3/32bit
CFLAGS= -bi -c -fc -g -apcs 3/32bit -processor ARM7TM -arch 4T

## >> When code down load to DRAM
ASMFLAG3        = -bi -apcs 3/32bit -g -PD "ROM_AT_ADDRESS_ZERO SETL {TRUE}"
LINKFLAG4       = -first init.o(Init) -RO 0x0 -RW 0x200000 -o diag.axf -aif -debug -Symbols diag.sym
$(OBJS) $(LIB)
OBJS        =  init.o start.o diag.o yours.o

diag.axf: $(OBJS)
    $(LINK) $(LINKFLAG4)
init.o: init.s
      $(ASM) $(ASMFLAG3)  init.s -o init.o -list init.lst
start.o: start.s
$(ASM) $(ASMFLAG1)  start.s -o start.o
diag.o: diag.c
$(CC) $(CFLAGS) -errors diag.err  diag.c
yours.o: yours.c
$(CC) $(CFLAGS) -errors yours.err  yours.c
```

After modifying makefile , type C:\> armmake diag.axf to get debugging image file.

Invoking ADW

If you modify SNDS100 board , ADW can be invoked without considering the elapsed time.

Setting the Special Registers of KS32C5000 series

Your write a text file to write special registers.

Your write a text file to write special registers.

When using EDO or Fast Page DRAM (armdram.ini at c root directory)

```
let 0x3ff3010 = 0xfffffffa
let 0x3ff3014 = 0x12040060
let 0x3ff3018 = 0x14042060
let 0x3ff301c = 0x16044060
let 0x3ff3020 = 0x18046060
let 0x3ff3024 = 0x1a048060
let 0x3ff3028 = 0x1c04a060
let 0x3ff302c = 0x8400039b
let 0x3ff3030 = 0x0601039b
let 0x3ff3034 = 0x0801439b
let 0x3ff3038 = 0x0a01839b
let 0x3ff303c = 0xce378360
```

When using Sync DRAM (armsdram.ini at c root directory)

```
let 0x3ff0000 = 0x87ffff91
let 0x3ff3010 = 0xfffffffa
let 0x3ff3014 = 0x12040060
let 0x3ff3018 = 0x14042060
let 0x3ff301c = 0x16044060
let 0x3ff3020 = 0x18046060
let 0x3ff3024 = 0x1a048060
let 0x3ff3028 = 0x1c04a060
let 0x3ff302c = 0x04000380
let 0x3ff3030 = 0x0601039b
let 0x3ff3034 = 0x0801439b
let 0x3ff3038 = 0x0a01839b
let 0x3ff303c = 0xce338360
```

After making text file , give a command (obey c:\armsdram.ini) at ADW command window. Then ADW initial KS32C5000 series system manager registers for interfacing with SDRAM. After this step , you have finished the preparation to download a image.

Selecting image file and Setting ADW for Debugging

Select File menu bar and Load Image menu and get your image file.Then type obey c:\armsd.ini (see Programmer' s Guide Section 2.).

Step into Debugging.

After doing all of previous work , you can run into debugging with code at DRAM based 0x0.