

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 debugging.

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 special 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.

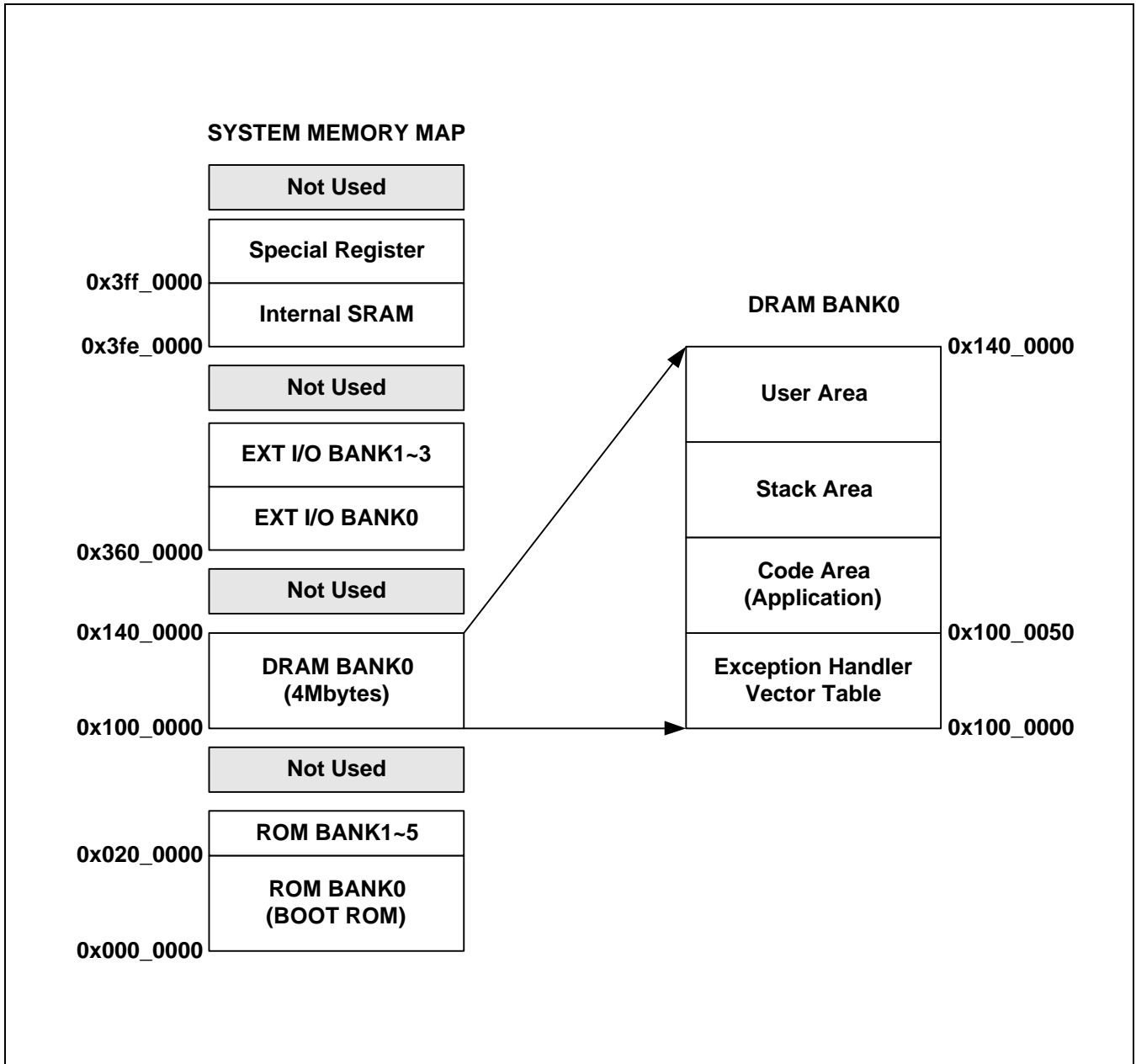


Figure 3-1. SNDS100 System Memory MAP

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.

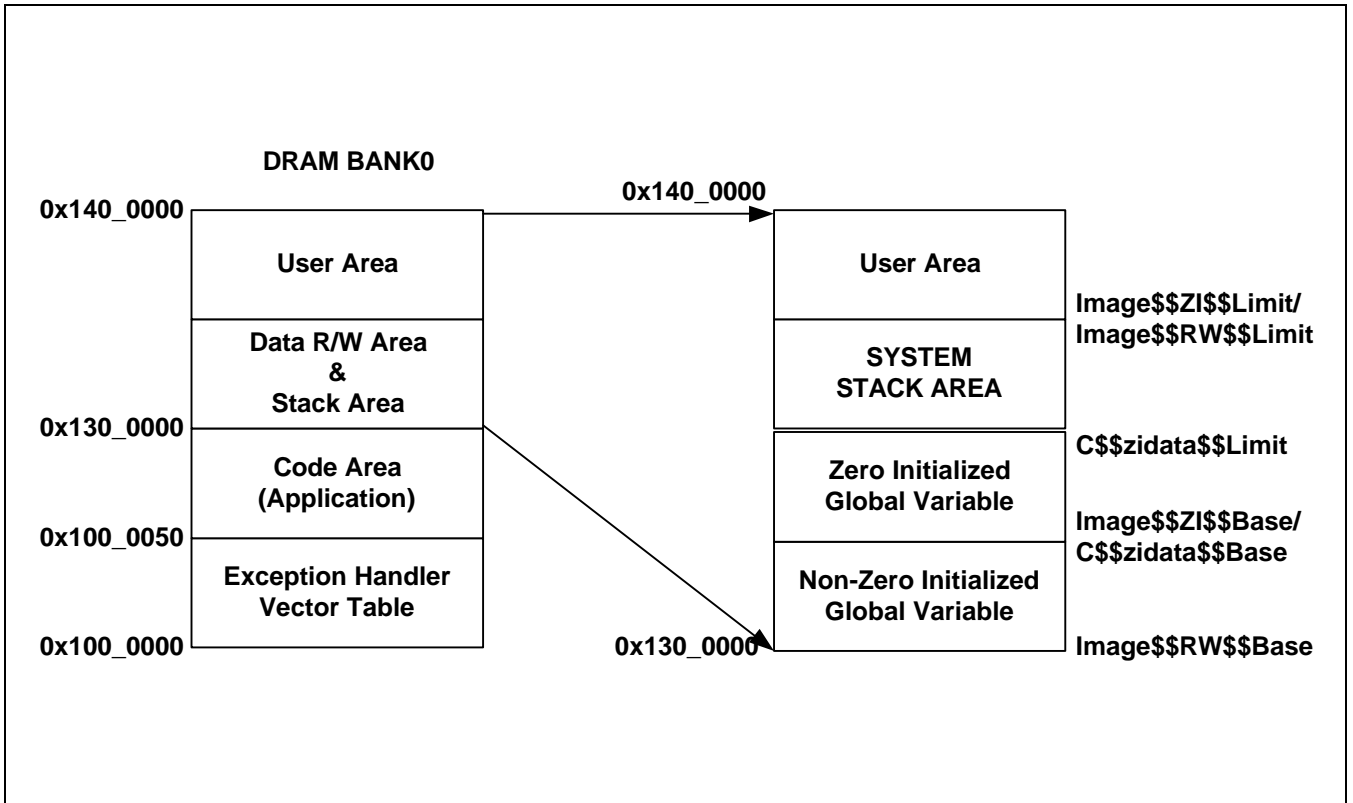


Figure 3-2. Data Structures of DRAM Bank0

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 programmed by assembler code and the diagnostic by C-Source code.

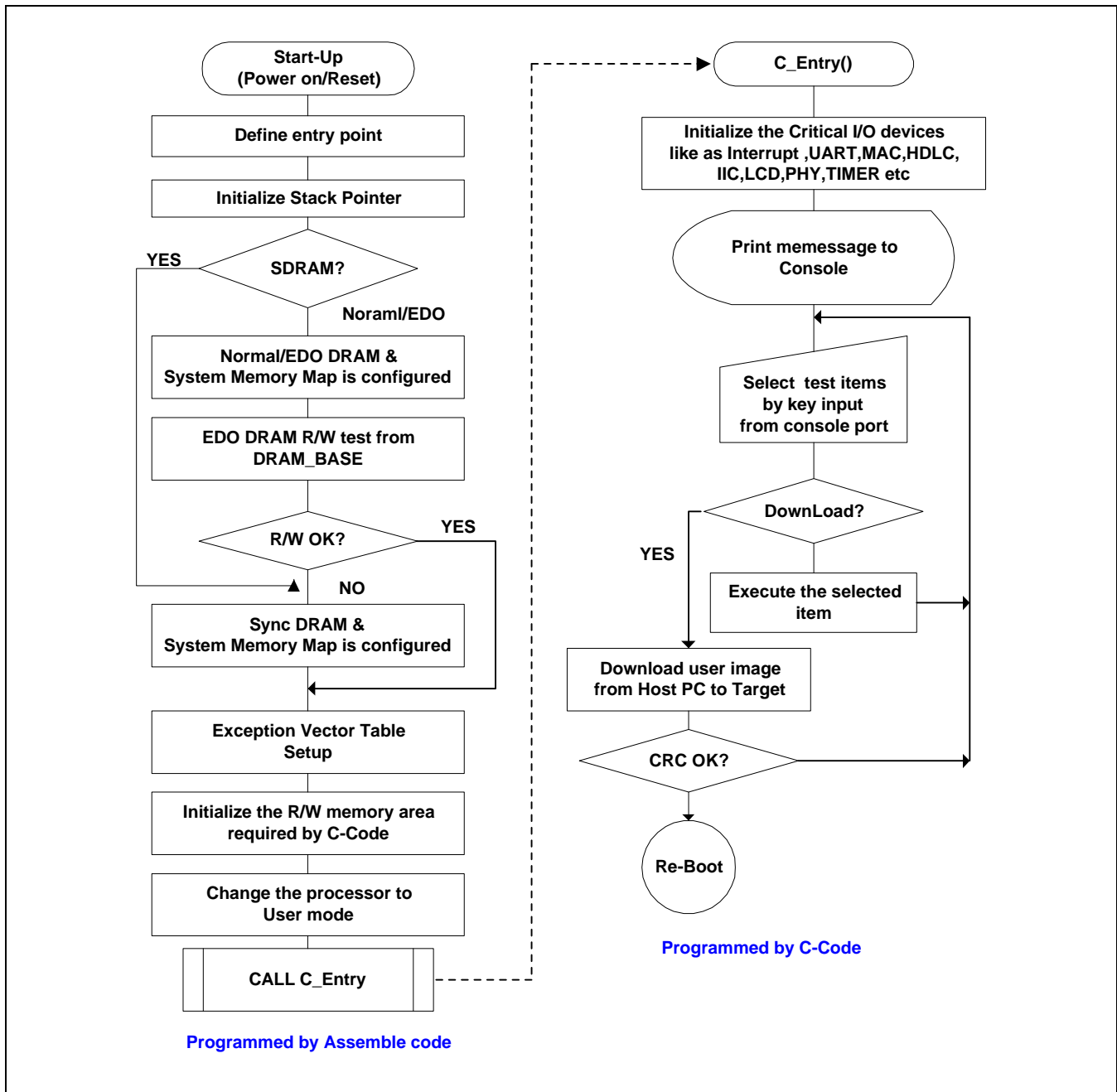


Figure 3-3. The Boot processing flows of Boot ROM.

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

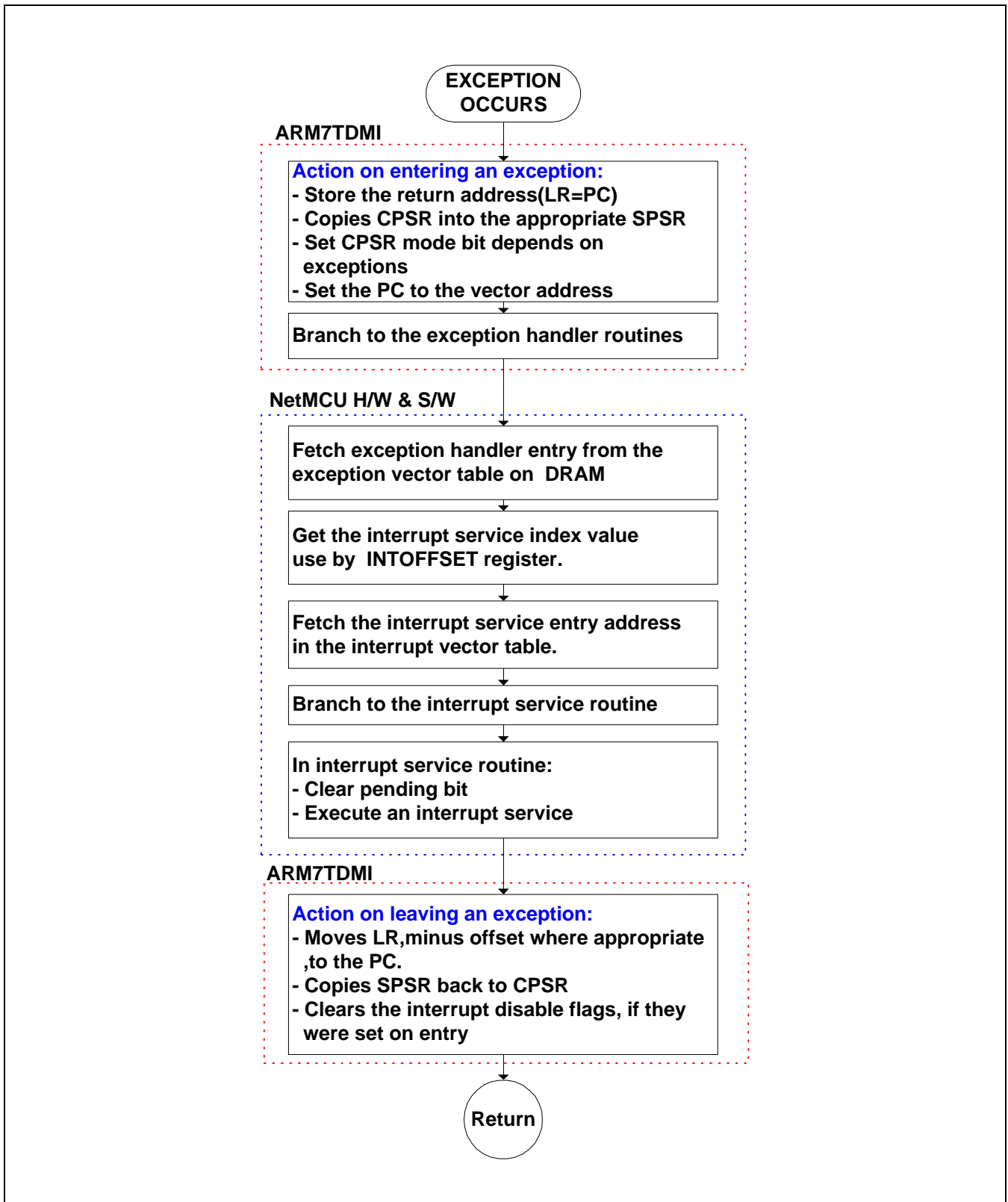


Figure 3-4. Exception handling flows of NetMCU devices

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
Sn ds.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.
Sn ds.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 initaliztion. 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 downloading 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 be 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 ; ROMCnt 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 ~ 0x01FFFFFF, ROM0,4Mbit,2cycle
    DCD rROMCON1       ;
    DCD rROMCON2       ; 0x0400000 ~ 0x05FFFFFF, ROM2
    DCD rROMCON3       ; 0x0600000 ~ 0x07FFFFFF, ROM3
    DCD rROMCON4       ; 0x0800000 ~ 0x09FFFFFF, ROM4
    DCD rROMCON5       ;
    DCD rDRAMCON0      ; 0x1000000 ~ 0x13FFFFFF, DRAM0 4M,
    DCD rDRAMCON1      ; 0x1400000 ~ 0x17FFFFFF, DRAM1 4M,
    DCD rDRAMCON2      ; 0x1800000 ~ 0x1EFFFFFF, DRAM2 16M
    DCD rDRAMCON3      ; 0x1C00000 ~ 0x1FFFFFFF
    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 ~ 0x01FFFFFF, ROM0,4Mbit,2cycle
    DCD rROMCON1       ;
    DCD rROMCON2       ; 0x0400000 ~ 0x05FFFFFF, ROM2
    DCD rROMCON3       ; 0x0600000 ~ 0x07FFFFFF, ROM3
    DCD rROMCON4       ; 0x0800000 ~ 0x09FFFFFF, ROM4
    DCD rROMCON5       ;
    DCD rSDRAMCON0     ; 0x1000000 ~ 0x13FFFFFF, DRAM0 4M,
    DCD rSDRAMCON1     ; 0x1400000 ~ 0x17FFFFFF, DRAM1 4M,
    DCD rSDRAMCON2     ; 0x1800000 ~ 0x1EFFFFFF, DRAM2 16M
    DCD rSDRAMCON3     ; 0x1C00000 ~ 0x1FFFFFFF
    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=4cycle
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=4cycle
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 here. 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.

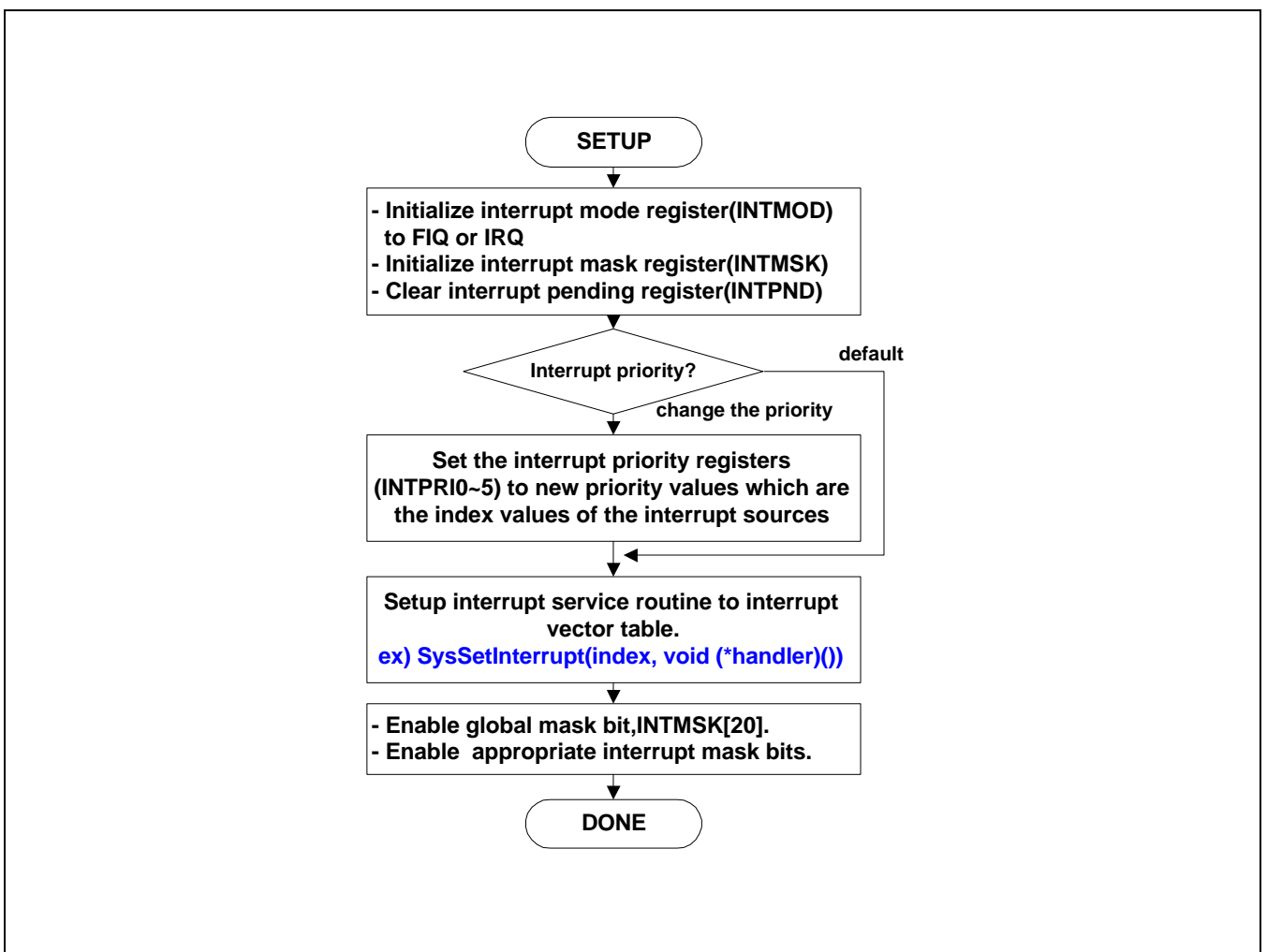


Figure 3-5. Setup concept diagrams for interrupt service routine

Listing 3-14. (isr.c)

```

void ClrIntStatus(void)
{
    INTMASK = 0x3ffff; // All interrupt disabled include global bit
    INTPEND = 0x1ffff; // All clear pending
    INTMODE = 0x1ffff; // 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
    INTPRIO = 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).

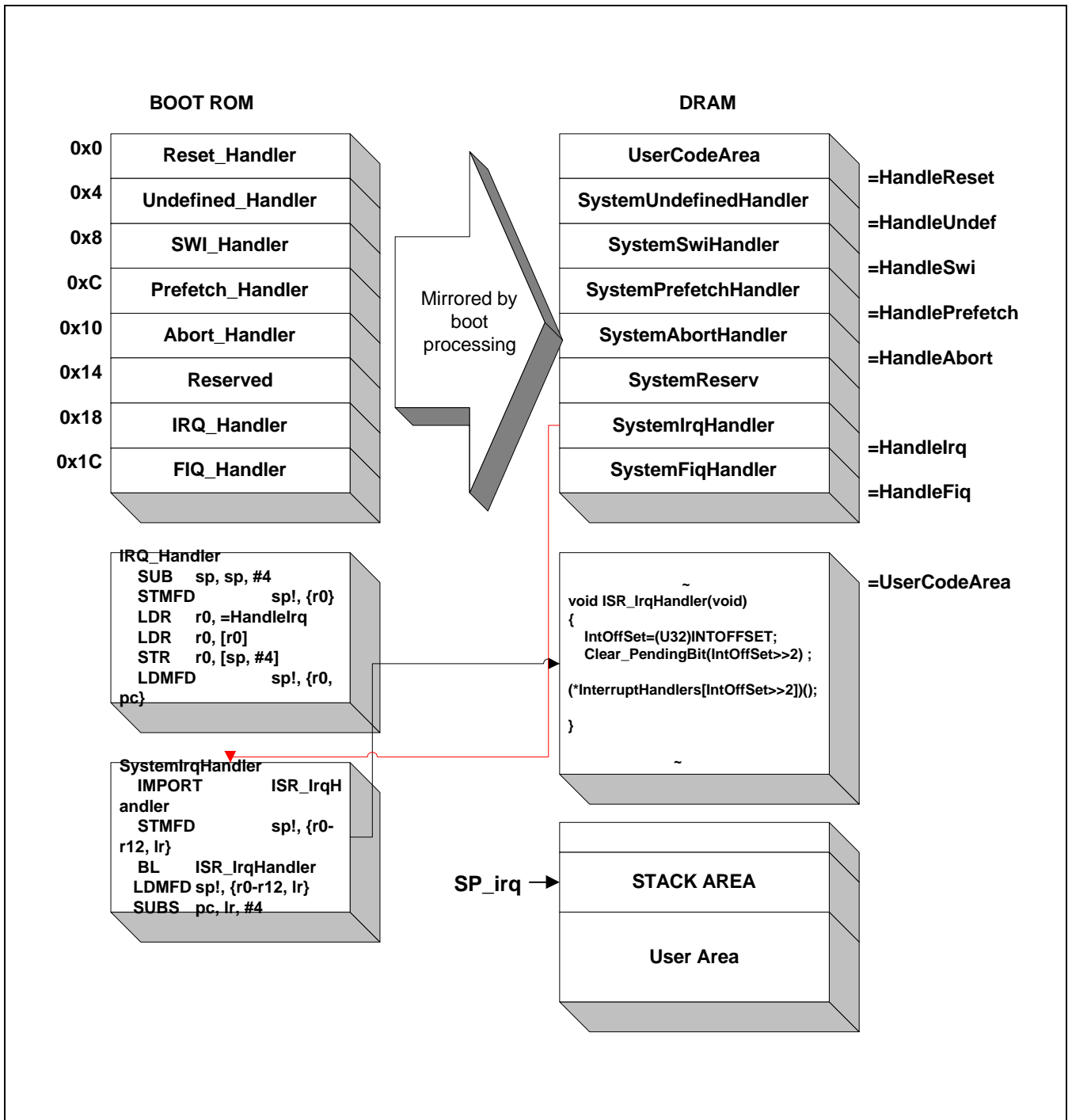


Figure 3-6. Example of IRQ interrupt handling process

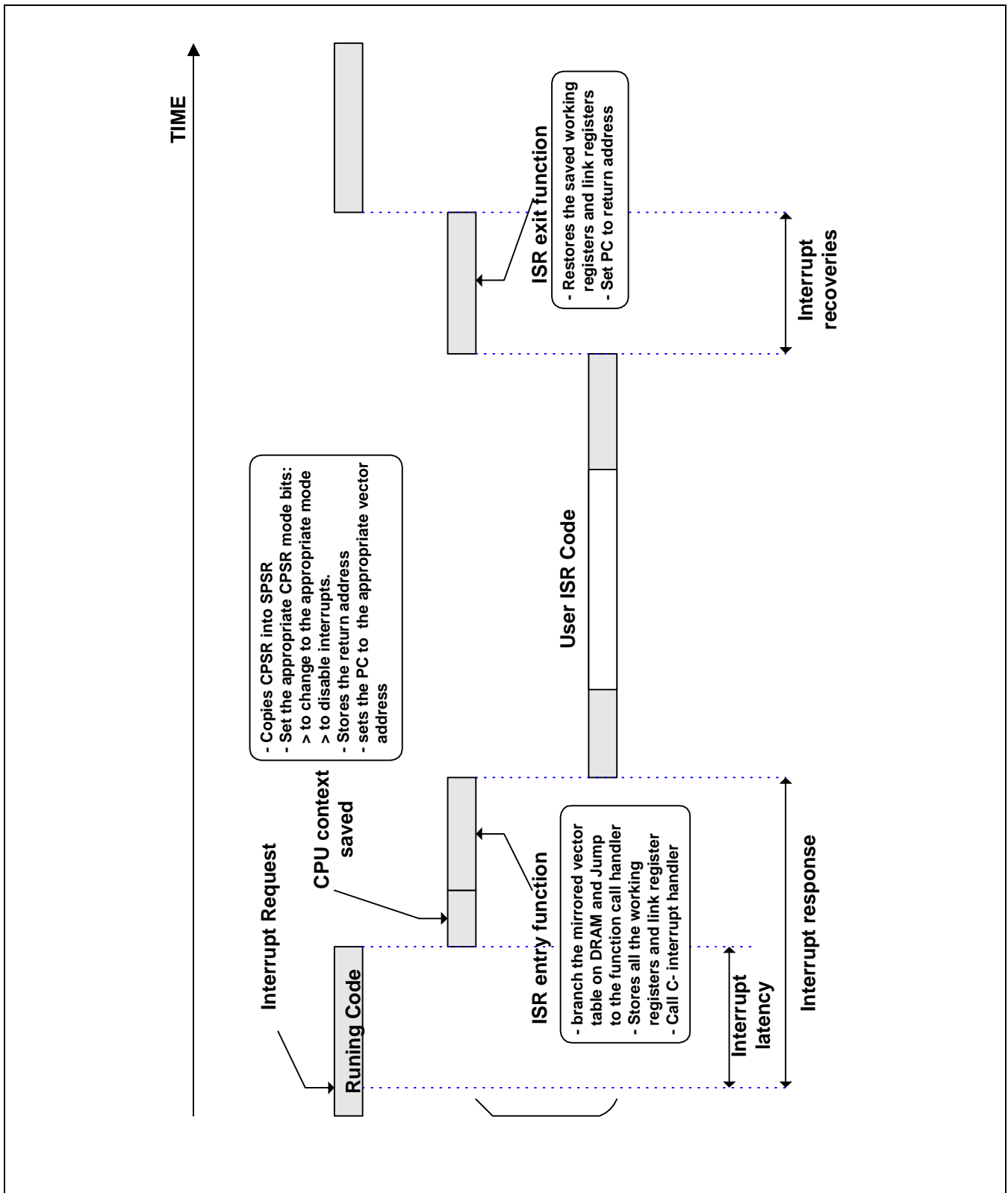


Figure 3-7. Interrupt Latency, Response and Recovers of NetMCU Device

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 ($T_{syncmax}$ if asynchronous), plus the time for the longest instruction to complete (T_{ldm} , the longest instruction is an LDM which loads all the registers including PC), plus the time for the data abort entry (T_{exe}), plus the time for FIQ entry (T_{fiq}). At the end of this time ARM7TDMI will be executing the instruction at 0x1C.

$T_{syncmax}$ is 3 processor cycles, T_{ldm} is 20 cycles, T_{exe} is 3 cycles, and T_{fiq} is 2 cycles. The total time is therefore 28 processor cycles.

Interrupt Latency = $T_{syncmax}$ (3cycles) + T_{ldm} (20cycles) + T_{exe} (3cycles) + T_{fiq} (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 ($T_{syncmin}$), plus T_{fiq} . 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

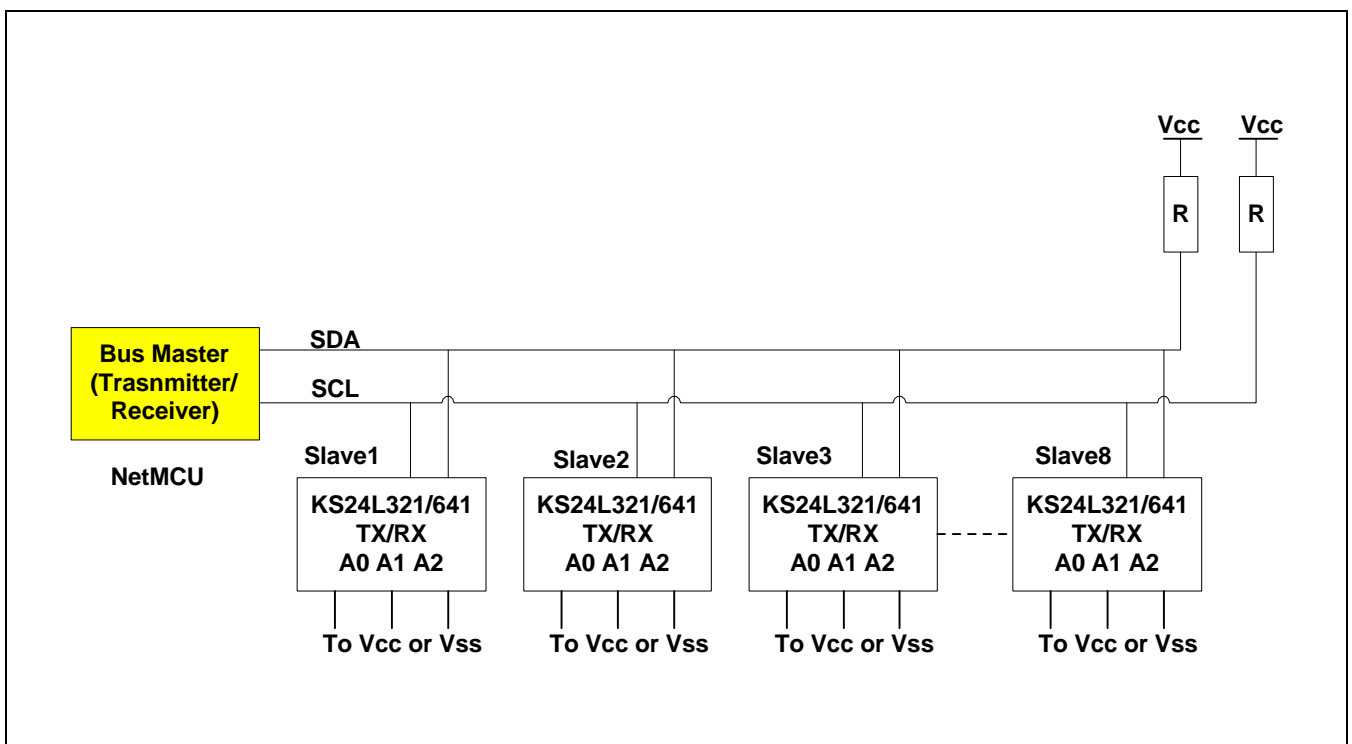


Figure 3-8. Typical Configuration

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).

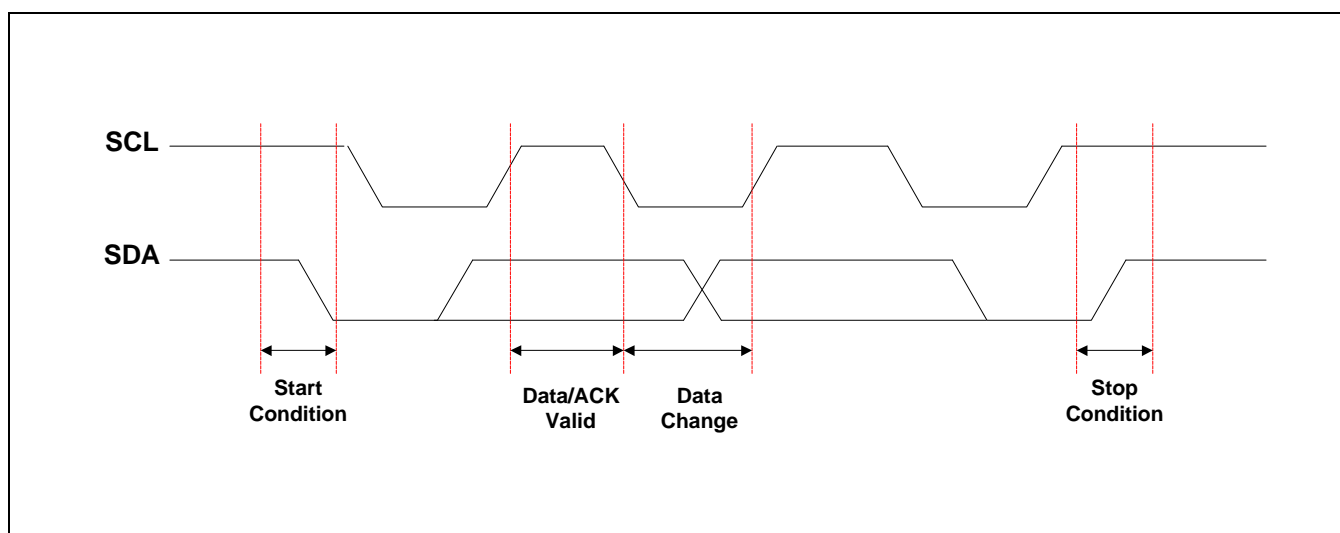


Figure 3-9. Data Transmission Sequence

- **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.

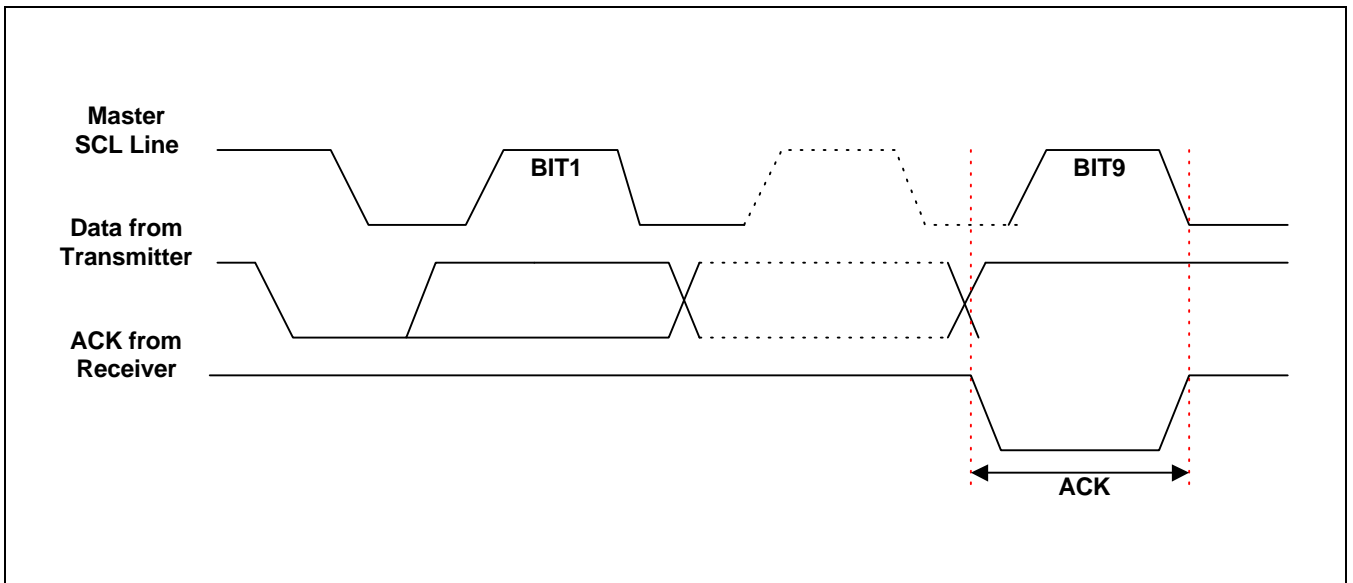


Figure 3-10. Acknowledge Response From Receiver

- **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

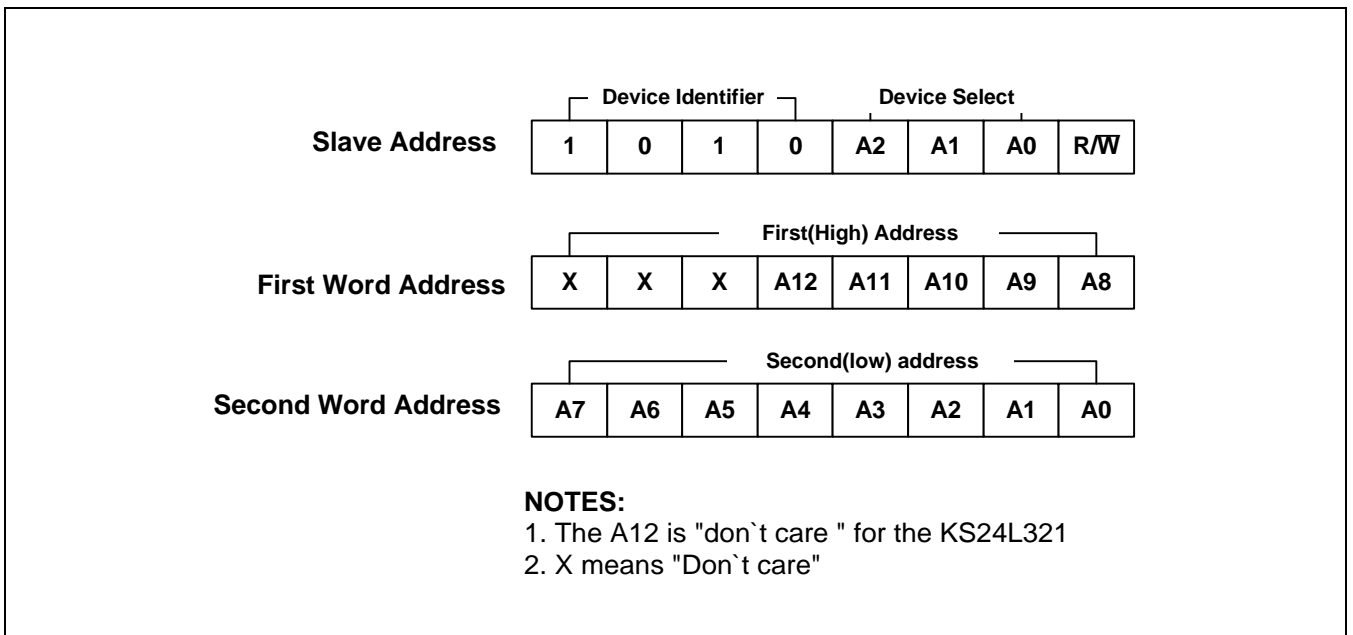


Figure 3-11. Device Address

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 licSetup(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).

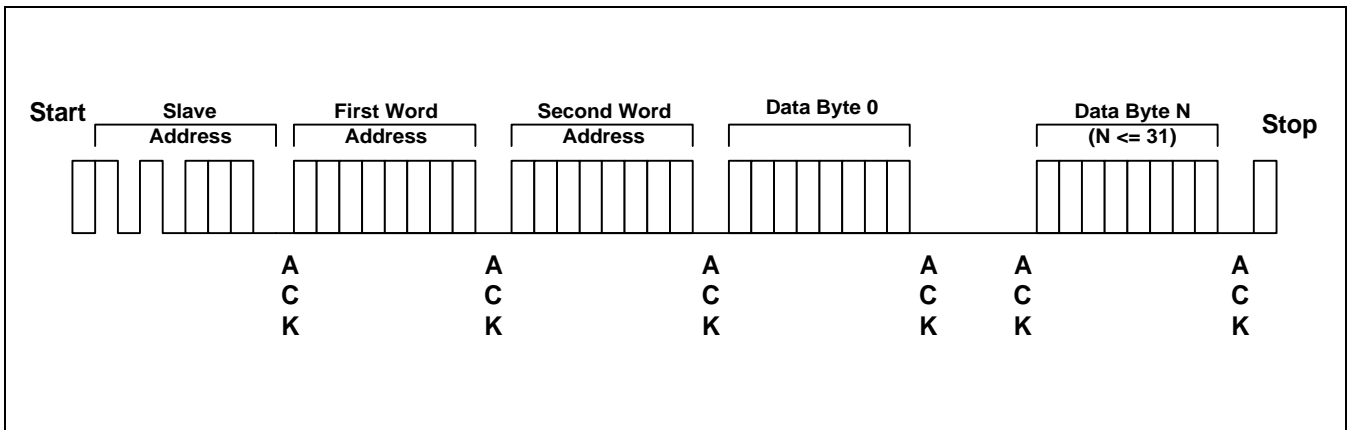


Figure 3-12. Page Write Operation

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,IICWriteIcr) ; /*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;
            }
        }
    }

    licSetup();
    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(IICON & BUSY); /* Wait! the iic bus is busy */
IICON = 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 IICWritelsr(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 */
        IICON = 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).

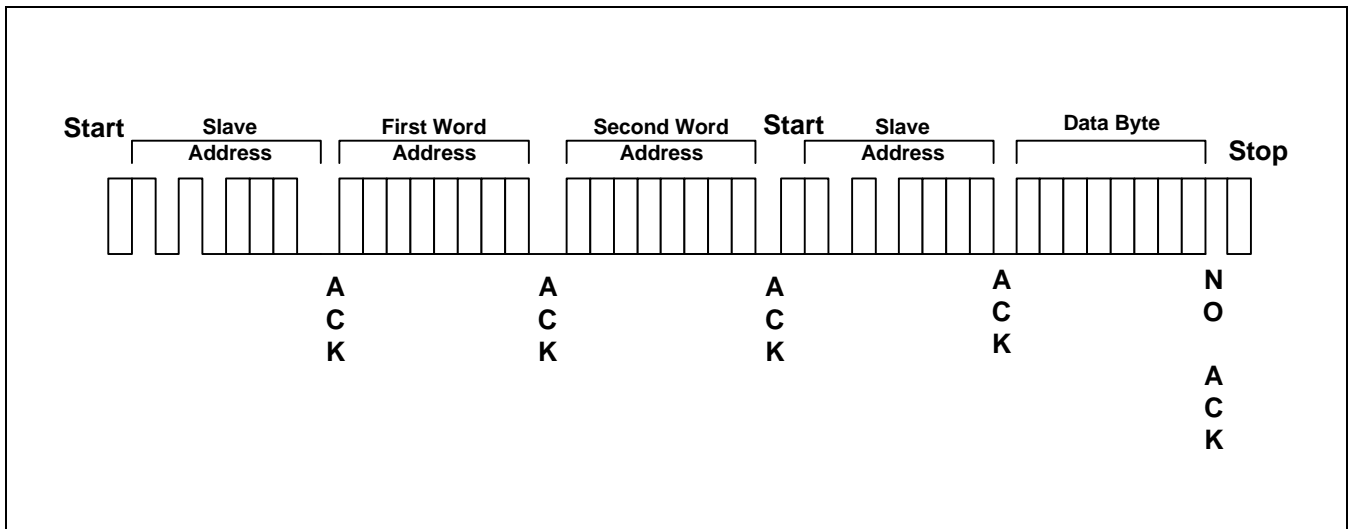


Figure 3-13. Random Address Byte Read Operation

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).

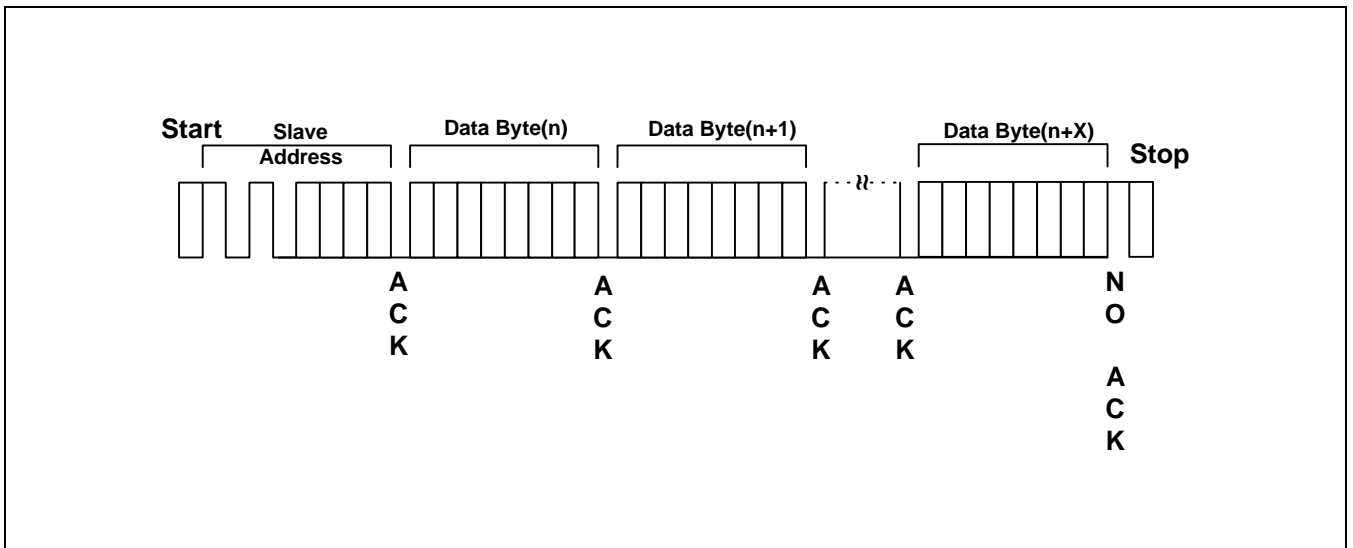


Figure 3-14. Sequential Read Operation

Listing 3-19. (iic.c)

```

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

    licSetup();
    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(IICON & BUSY); /* Wait! the iic bus is busy */
    IICON = 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 */
        IICON = RESTART;
        IICON = 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 ASYNCHRONOUS 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);
    UARTRxIntOff(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\rString echo for interrupt test.");
        i_printf("\rInput characters will be echoed on Console");
        i_printf("\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.

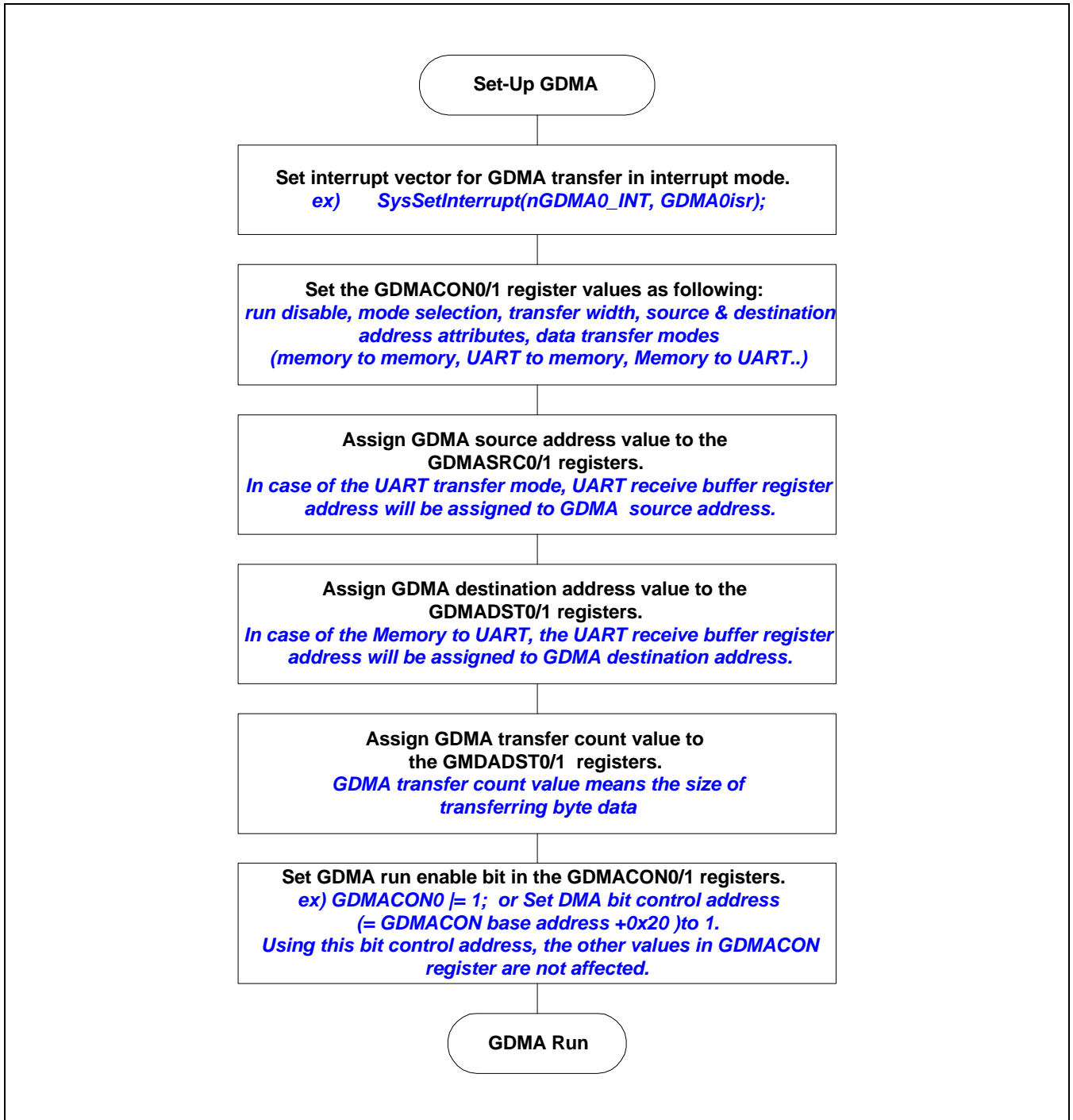


Figure 3-15. Concept Diagram for Setting Up GDMA

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

    GDMA_SRC0 = dmasrc;
    GDMA_DST0 = dmadst;
    GDMA_CNT0 = Size;

    DMA_CON_SET = GDMA_MEM2MEM ;

    switch(Width) {
    case 0 : DMA_CON_SET |= GDMA_WIDTH_BYTE; 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 */
    GDMA_CON0 = DMA_CON_SET | GDMA_RUN; /* Start run GDMA0 */

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

/* GDMA0 Transfer done interrupt 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.

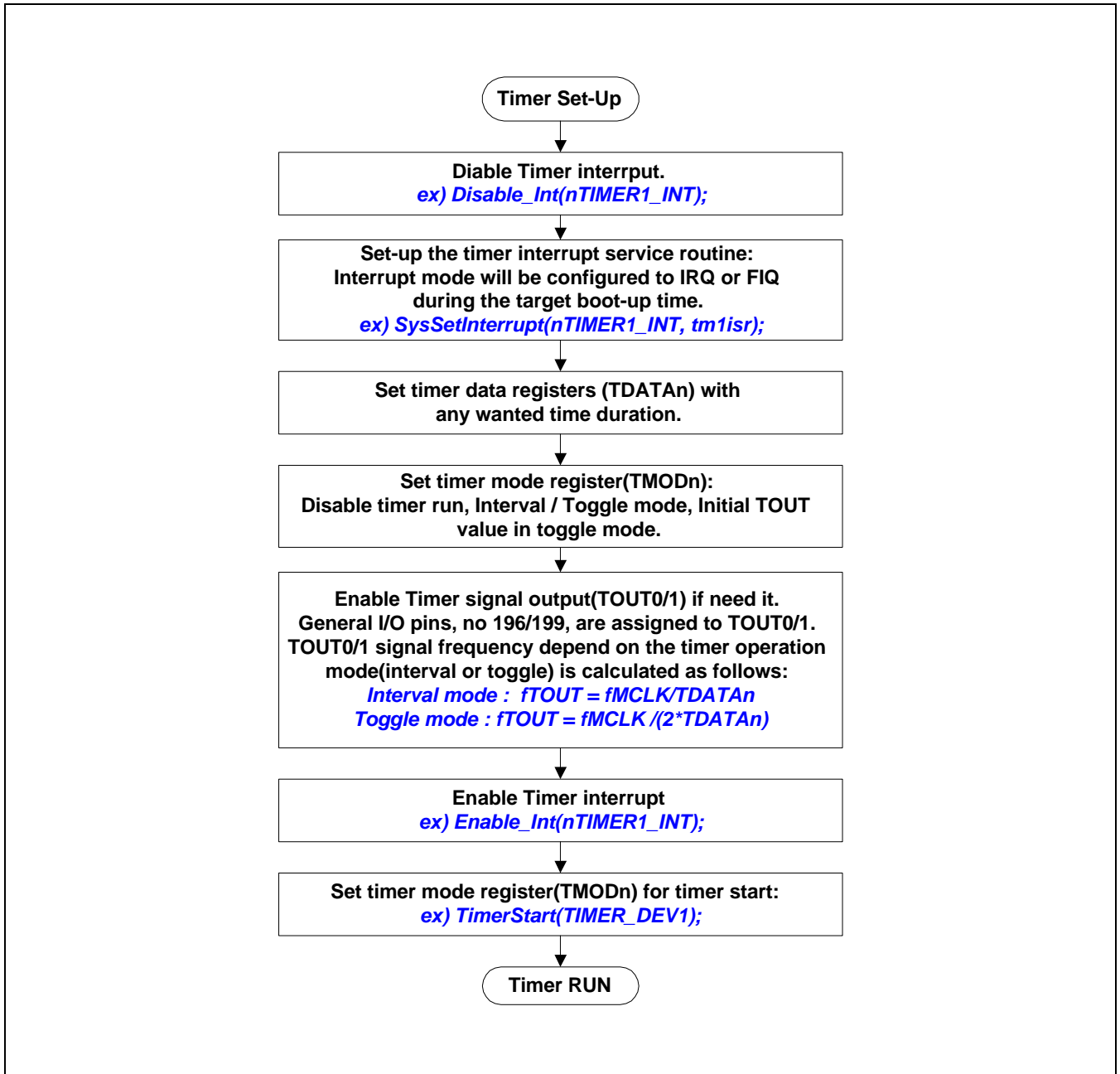


Figure 3-16. Concept Diagram for TIMER Set-Up.

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.

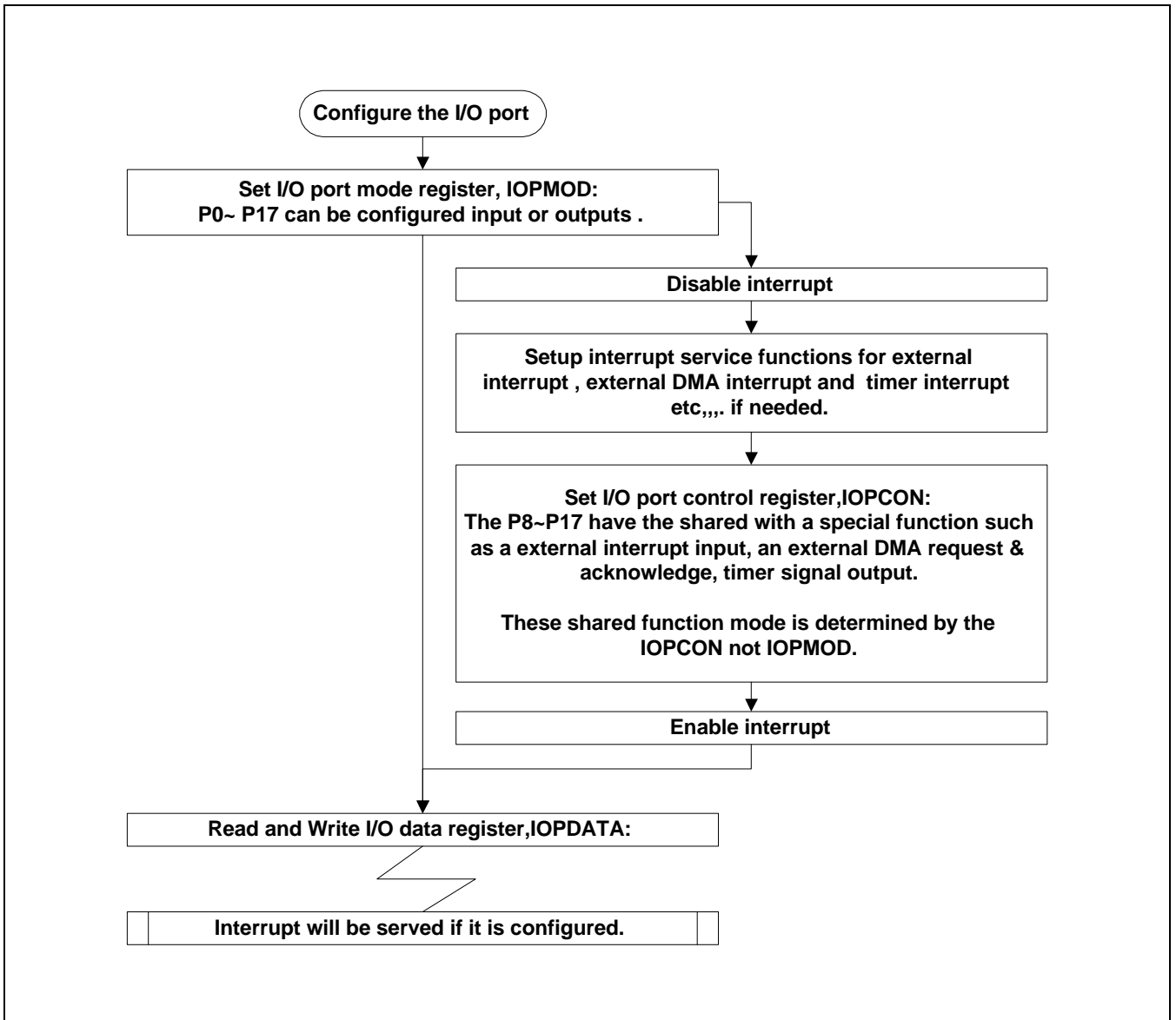


Figure 3-17. Concept Diagram for the Configuring of I/O Port

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) (HDLGINIT.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 Generator 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 ;
}
```

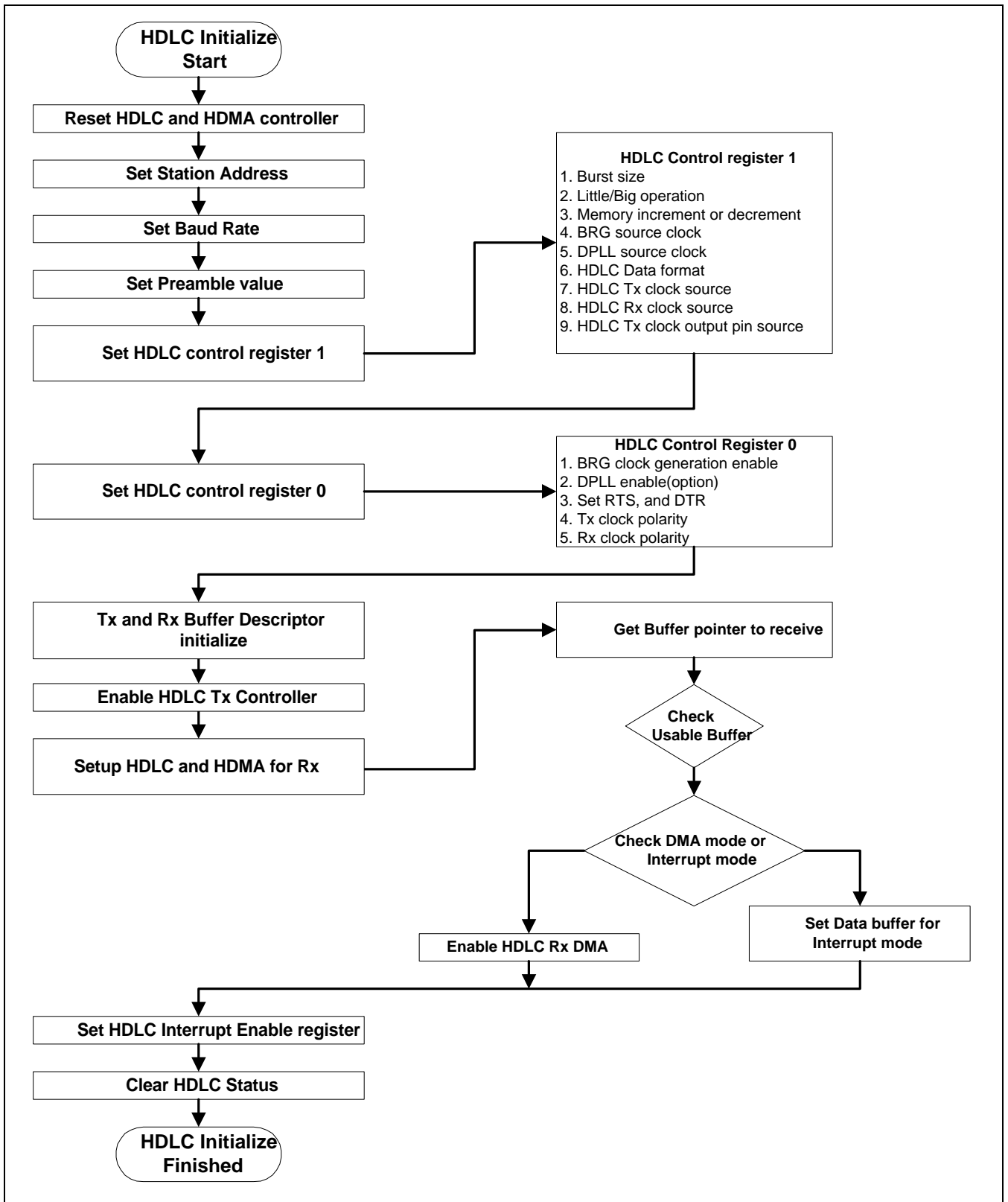


Figure 3-19. HDLC Initialization flow

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 Generator 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 Manchester, 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**.

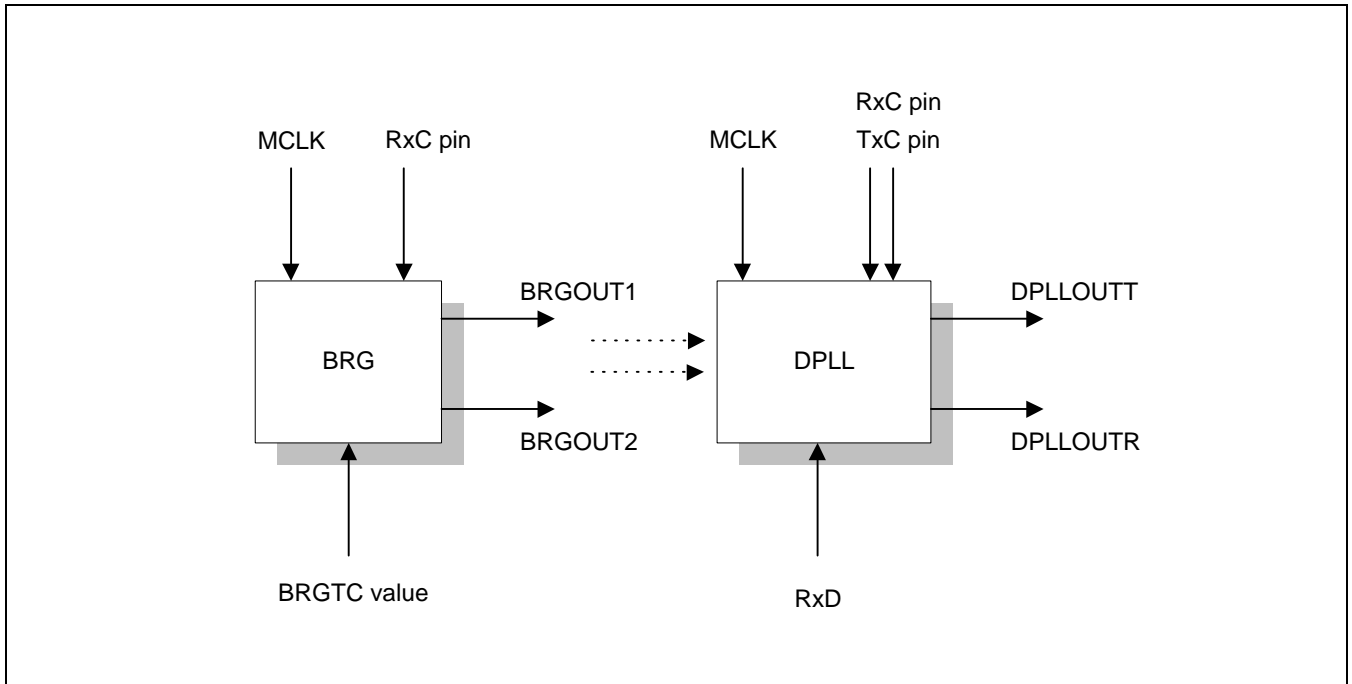


Figure 3-20. HDLC Internal clock Block Diagram

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**.

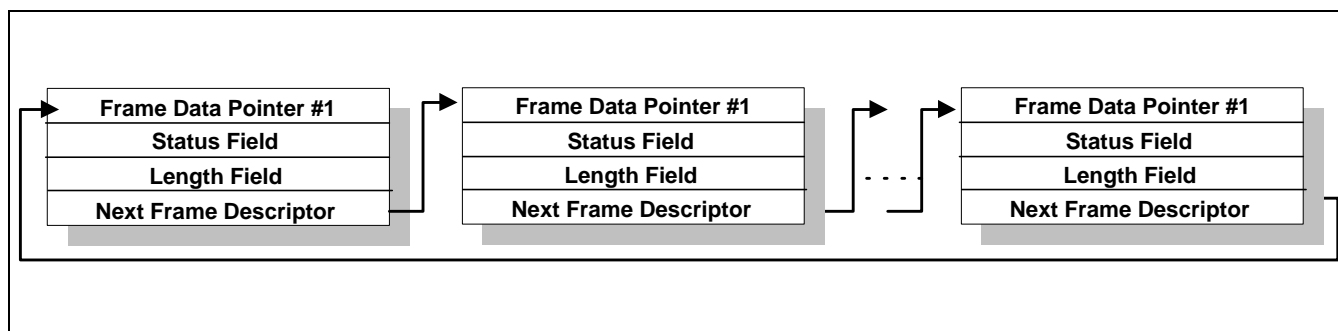


Figure 3-21. HDLC Buffer Descriptor Structure for KS32C5000(A)

Listing 3-34. HDLC Buffer descriptor structure for KS32C5000(A) (HDLINIT.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 incoming 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 {
            Modelnt_RxDataBuffer[channel] = (U32 *)DataBuffer ;
            Modelnt_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, Tx FIFO frame continue register, and at the end of transmit CPU write last frame data to HTXFIFOT, Tx FIFO 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) (HDLCLINIT.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
        Modelnt_TxDataBuffer[channel] = (U8 *)DataBuffer ;
        Modelnt_TxDataSize[channel] = Size ;

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

    return 1 ;
}

```

Listing 3-40. WriteDataToFifo() function for KS32C5000(A) (HDLINIT.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 ( ModelInt_TxDataSize[channel] > 4 ) {
            // Read data buffer pointer to transmit
            Data = (U32 *)ModelInt_TxDataBuffer[channel] ;

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

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

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

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

                // process under 4 byte data
                for ( j=0 ; j<4; j++) {
                    ModelInt_TxDataSize[channel] -- ; // decrement byte
                    if (ModelInt_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  
}
```

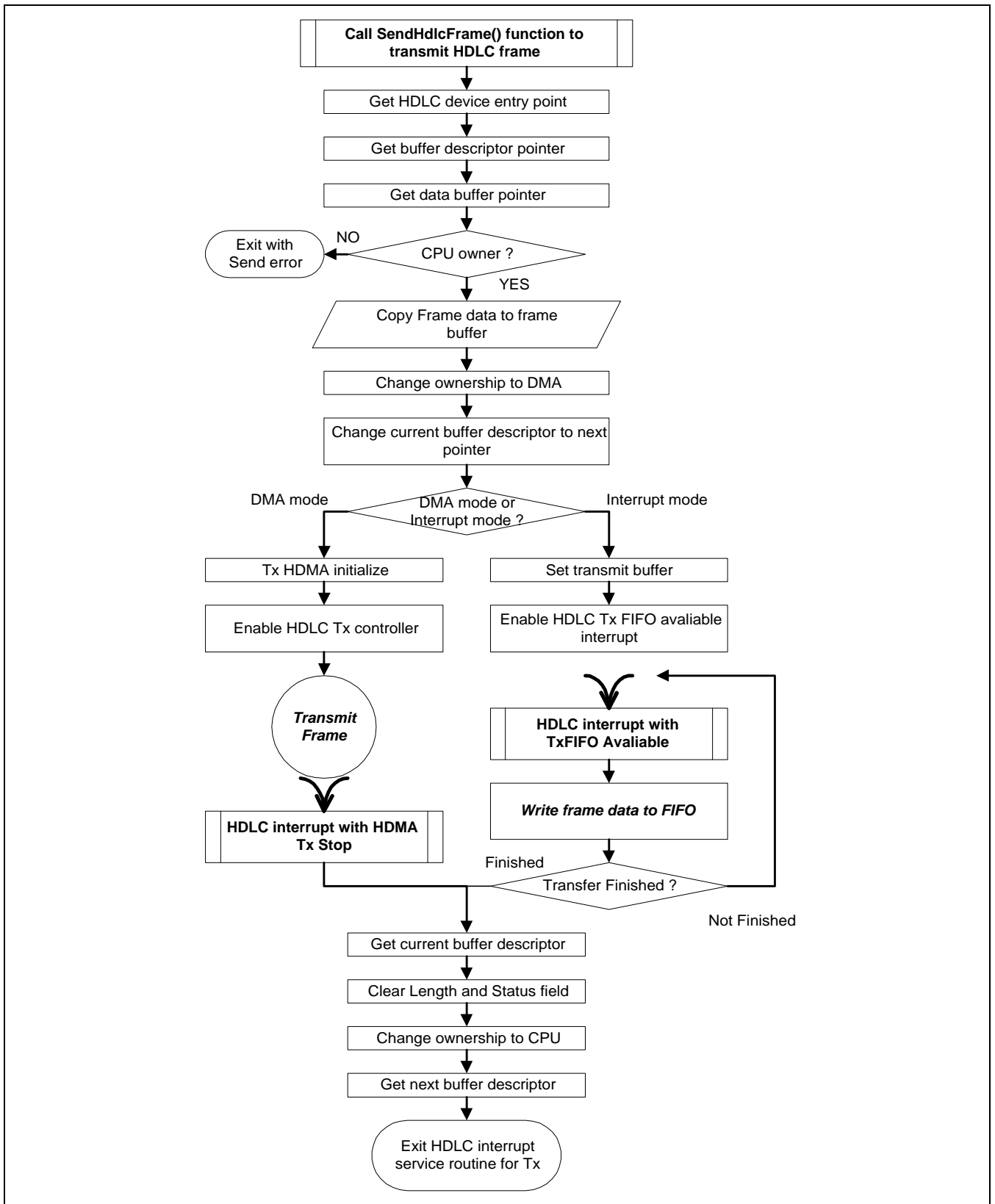



Figure 3-22. HDLC Transmit Frame Data Flow for KS32C5000(A)

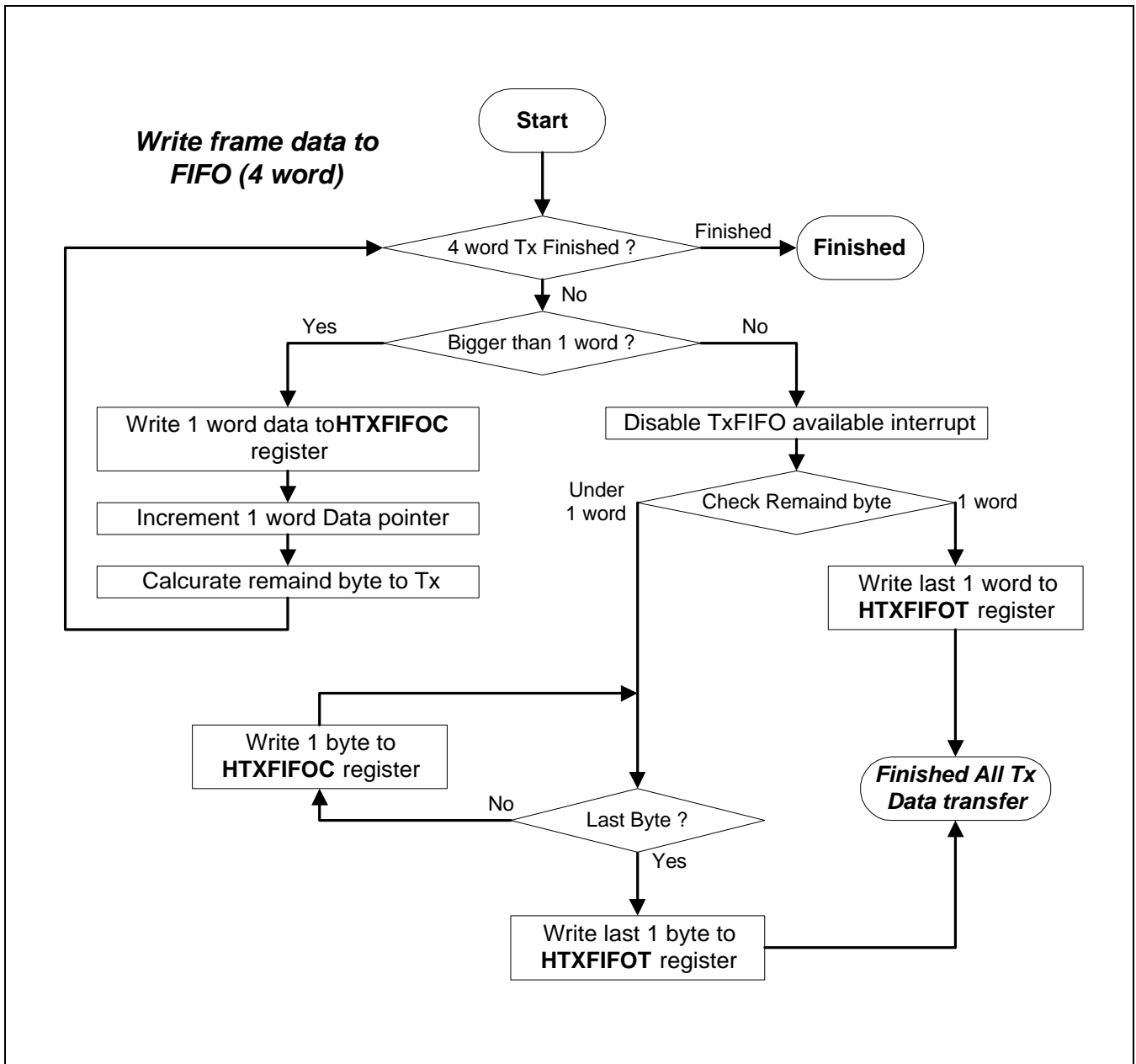


Figure 3-23. HDLC Tx FIFO Write Flow for KS32C5000(A)

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 ( ModelInt_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 Rx FIFO. 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].RxFIFOAvailable++;

    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 =
            ModelInt_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**.

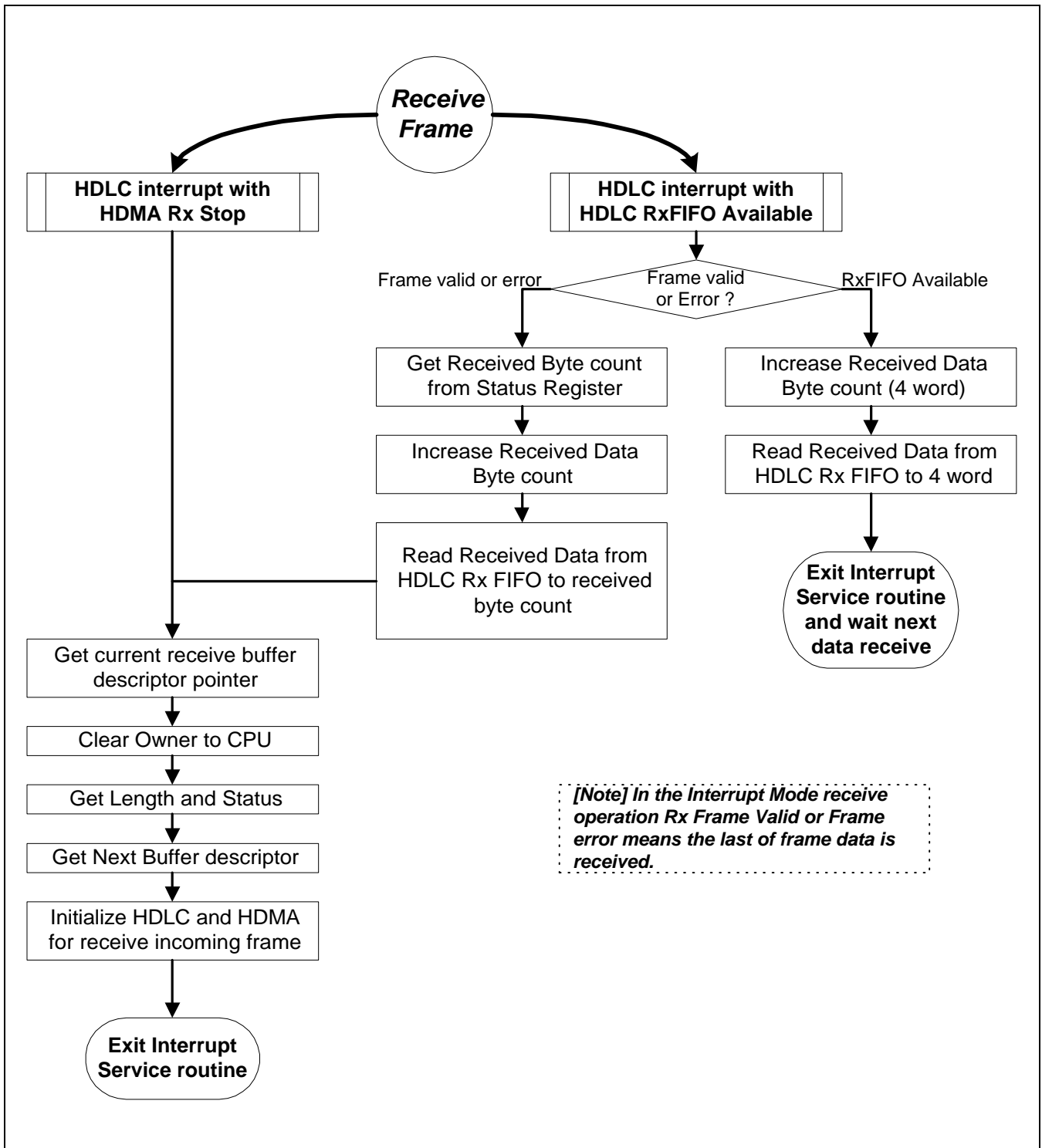


Figure 3-24. HDLC Receive Flow for KS32C5000(A)

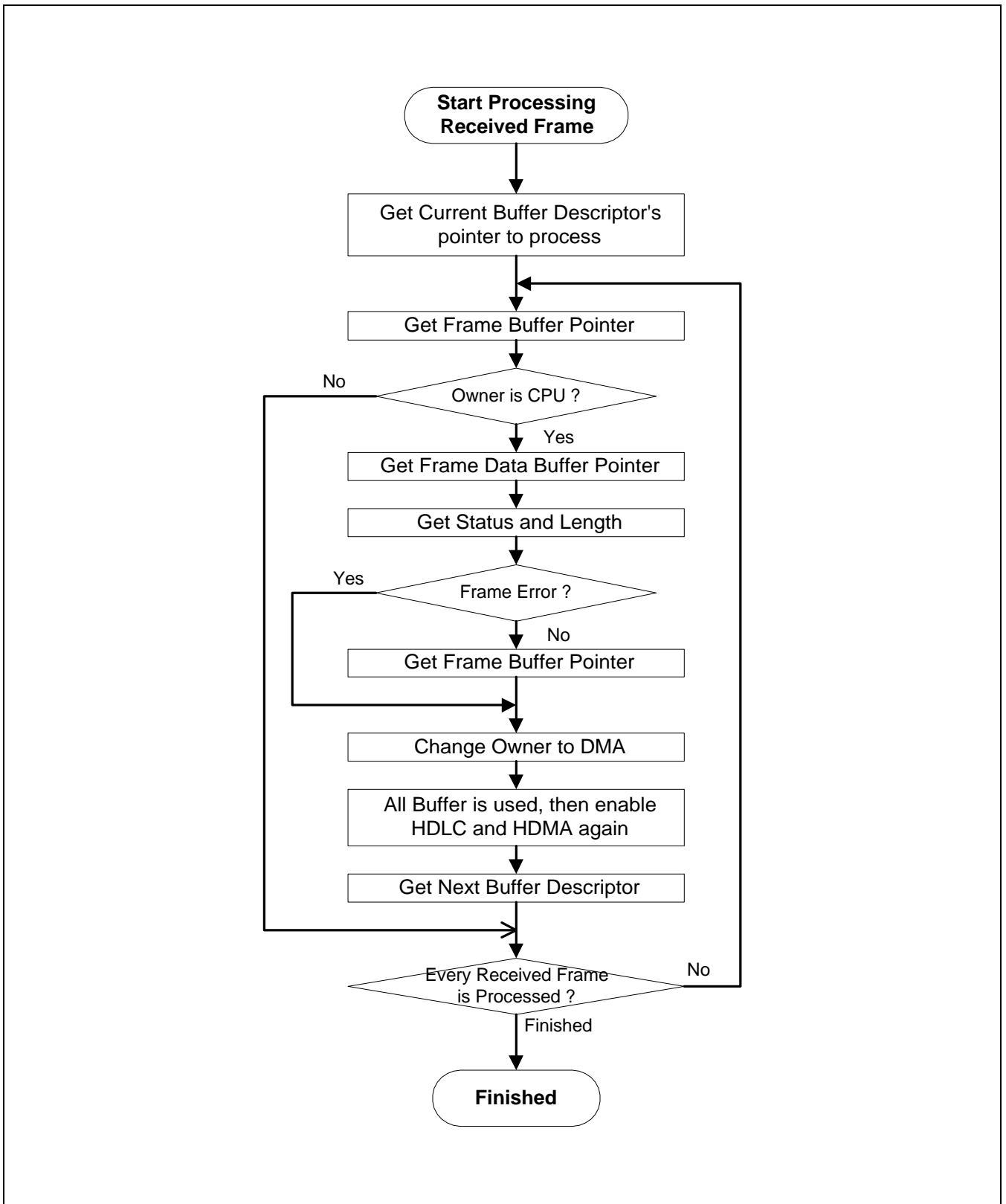


Figure 3-25. HDLC Processing Received Frame Flow for KS32C5000(A)

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) |= DPLEN|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

}
    
```

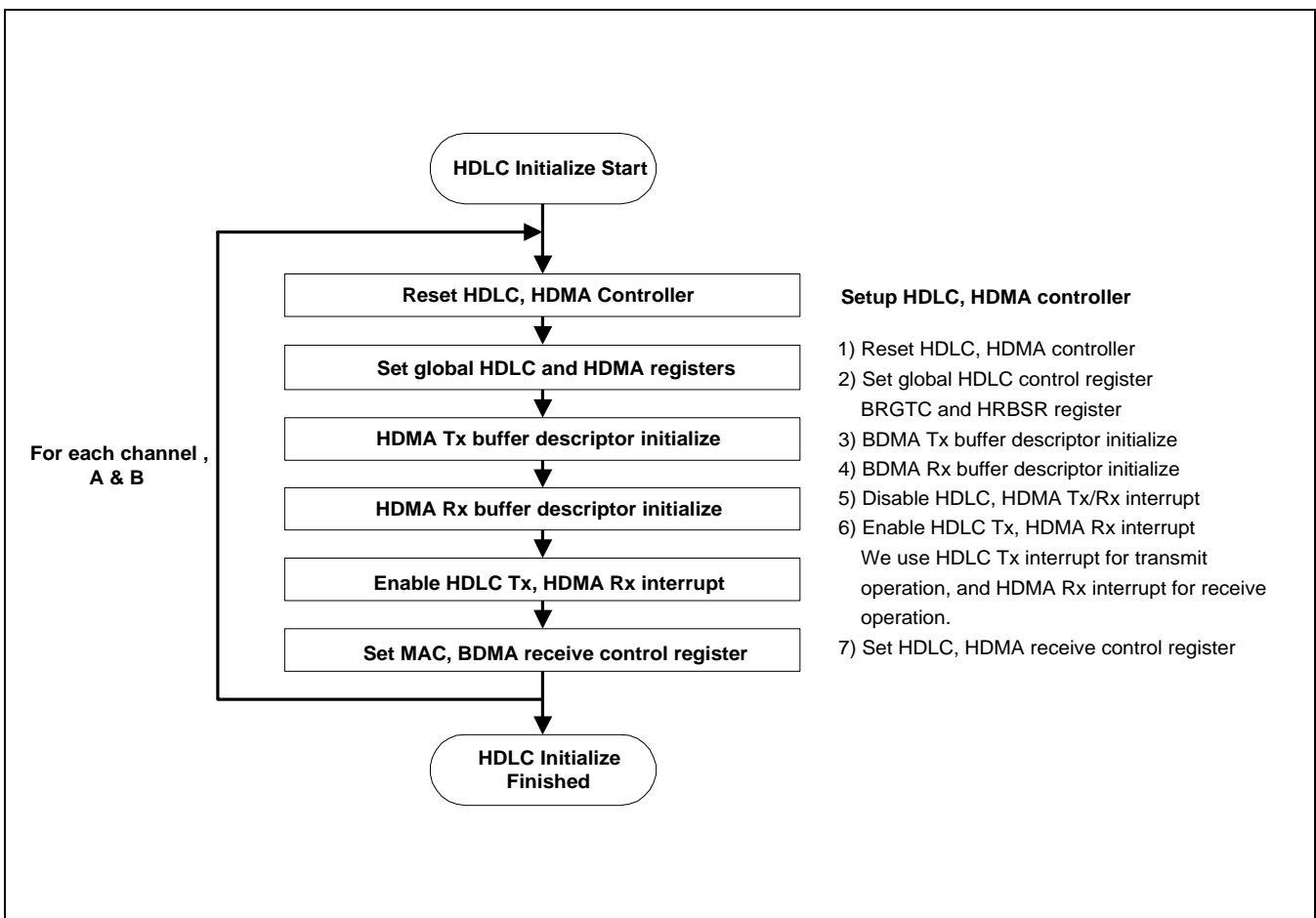


Figure 3-26. HDLC Initialize flow

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.

$$\text{BRGOUT1} = (\text{MCLK2 or RxC}) / (\text{CNT0} + 1) / (16^{\text{CNT1}})$$

$$\text{BRGOUT2} = \text{BRGOUT1} / (1 \text{ or } 16 \text{ or } 32 \text{ according to CNT2 value of the HBRGTC})$$

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

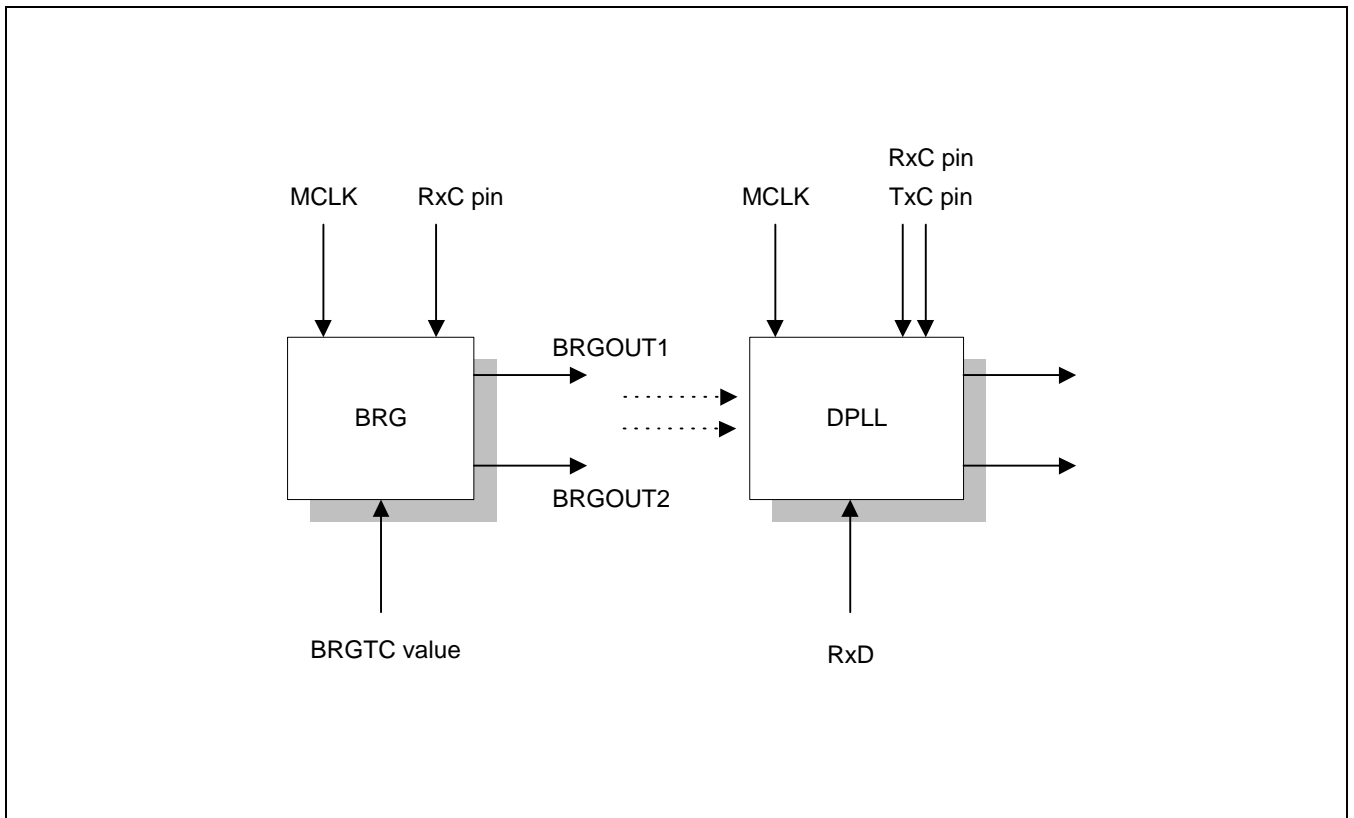


Figure 3-27. HDLC Internal clock Block Diagram

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) = 0xffff0000;
}
```

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;
```

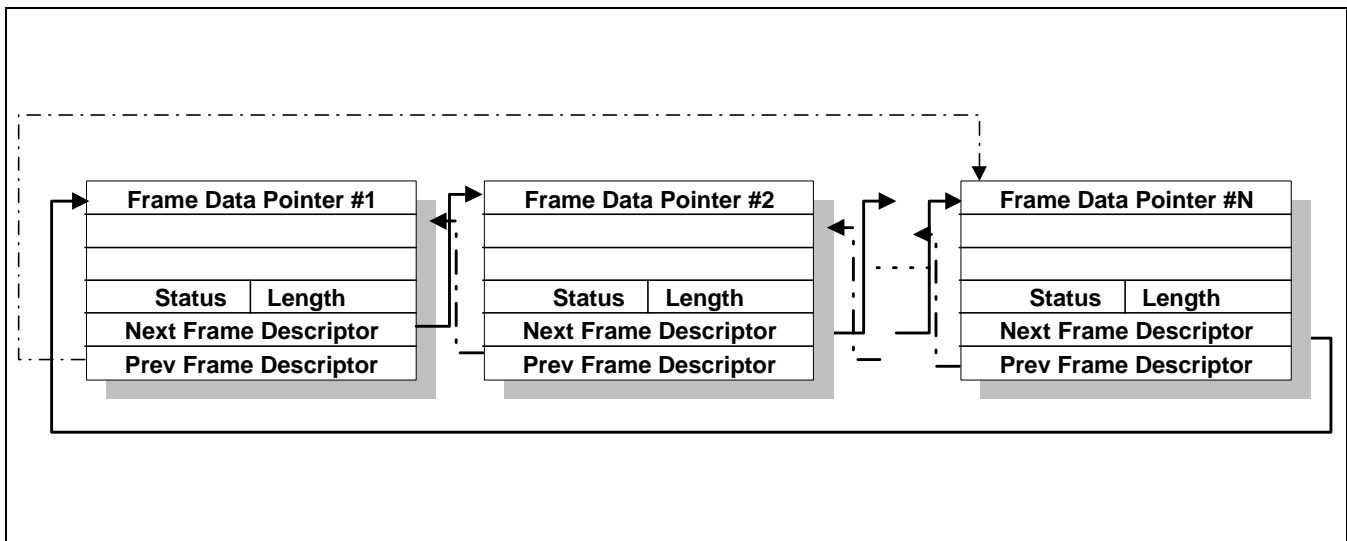


Figure 3-28. HDMA Buffer Descriptor Structure

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, using 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                    TxBDDataStartAddr ;
    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 ;

    TxBDDataStartAddr     = (U32)TxBBaseAddr ;
    TxBDDataStartAddr    |= 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)TxBDDataStartAddr & 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 ;
    }
}

```

```

        TxBDDataStartAddr += 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 RxBDDataStartAddr ;
    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 ;
    RxBDDataStartAddr = (U32)RxBBaseAddr[channel] | NonCache ;

    for(i=1; i<=MaxRxBDCount; i++){
        psRxBD->BufferDataPtr = (U32)RxBDDataStartAddr | 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 ;

        RxBDDataStartAddr += 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) TxBBBaseAddr | 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 details 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 available	When DMA mode , always ' 0'	I	After writing data into Tx FIFO	Yes
RxFA	(None)	(None)	When RxFIFO is available	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	automatically	No
TxSCTS	(None)	(None)	When nCTS transit	Indicate nCTS pin transition occurred	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 successfully.	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

RxD CD	(None)	(None)	When nDCD pin is low	Indicate nDCD pin state	D & I	automatically	No
RxSD CD	(None)	(None)	When nDCD transit	Indicate nDCD pin transition occurred	D & I	writing ' 1'	Yes
RxFV	(None)	(None)	the last byte of frame is transferred into the last location of the Rx FIFO 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
DPLL OM	(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
DPLL TM	(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 recommend you discard frame.		writing ' 1'	Yes
RxCRC E	(None)	RxIERR	frame is completed with CRC error		D & I	automatically	Yes
RxNO	(None)	(None)	received data is non-octet aligned frame		D & I	automatically	Yes
RxOV	(None)	(None)	the received data is transferred into the Rx FIFO when it is full.	A loss of data . continued overruns destroy data in the first FIFO register	D & I	automatically	Yes
RxMOV	(None)	(None)	no more buffer during receiving data	DRxEN is cleared.	D & I	writing ' 1'	Yes
DRx FD	(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 Rx FIFO 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 Length 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[HDLC_A].DMATxFD++ ;
        HSTATA = IntHDLCStatus & DTxFD ;
    }

    CheckStatus = TxSCTS|TxU|DTxABT|DTxNL|DTxNO ;
    if(IntHDLCStatus & CheckStatus){
        if(IntHDLCStatus & (DTxNO|DTxABT|DTxNL)){
            if(IntHDLCStatus & DTxABT){
                gHTxStatus[HDLC_A].DMATxABT++ ;
                HSTATA = IntHDLCStatus & DTxABT ;
            }
            if(IntHDLCStatus & DTxNO){
                gHTxStatus[HDLC_A].DMATxNO++ ;
                HSTATA = IntHDLCStatus & DTxNO ;
            }
            if(IntHDLCStatus & DTxNL){
                gHTxStatus[HDLC_A].DMATxNL++ ;
                HSTATA = IntHDLCStatus & DTxNL ;
            }
        }
    }
    if(!CheckTxStatus(HDLC_A))        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**.

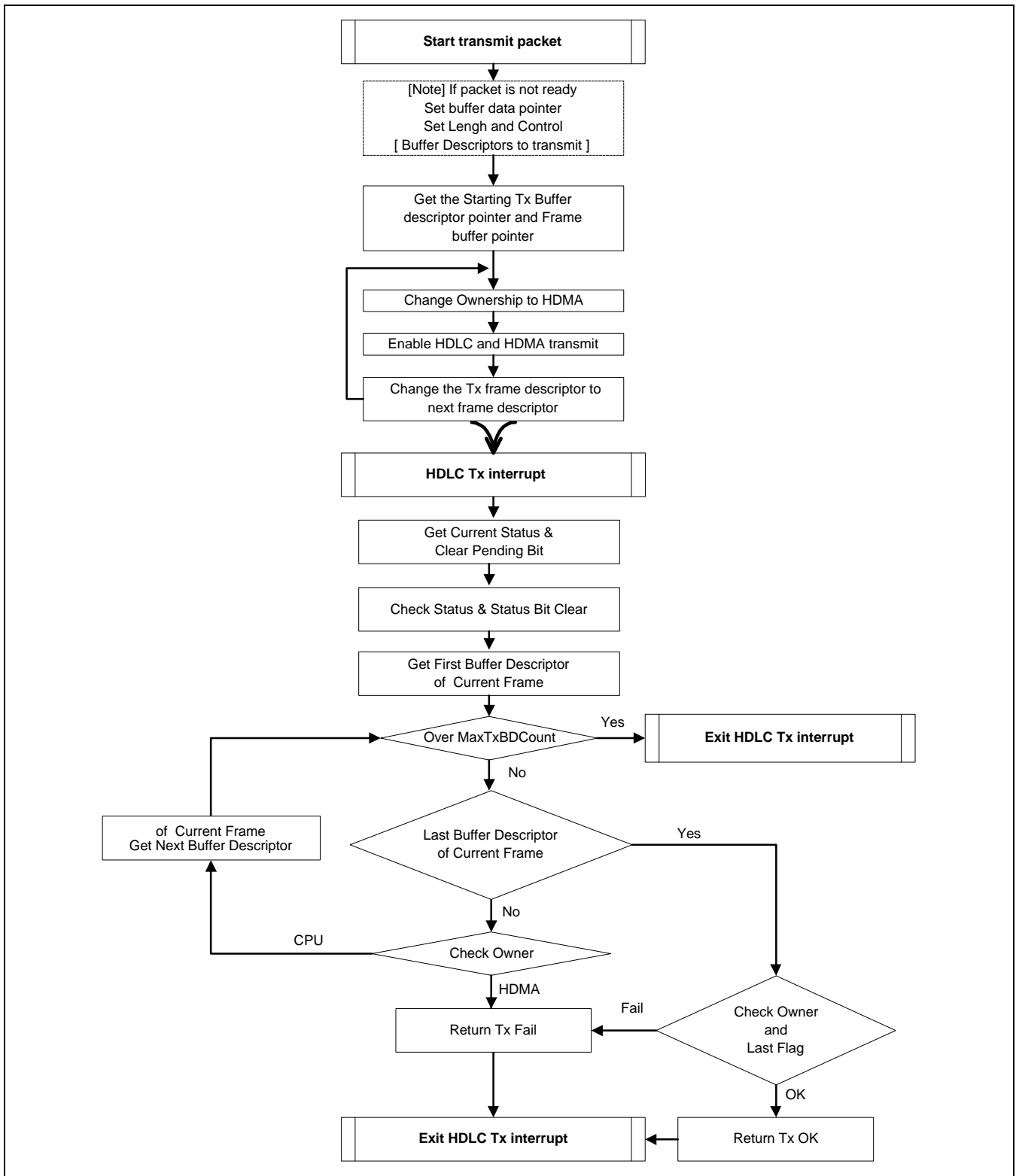


Figure 3-29. HDLC Frame transmit flow

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 occurred , 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 (HDLCAINIT.C)

```

/*
 * Function : HDMA_Rx_isr
 * Description : Interrupt Service Routine for HDMA Rx
 * Ethernet Frame is received in HDMA_Rx_isr
 */
void HDLCRxA_isr(void)
{
    U32 IntHDLCStatus, pRxBDPtr ;
    U32 RemainByte ;
    IntHDLCStatus = HSTATATA ;
    Clear_PendingBit(nHDLCRxA_INT) ;

    if(IntHDLCStatus & DRxFD){
        gHRxStatus[HDLC_A].DMARxFD++ ;
        HSTATATA = IntHDLCStatus & DRxFD ;

        if(IntHDLCStatus & RxCRCE) {
            gHRxStatus[HDLC_A].RxCRCErr++ ;
            HSTATATA = IntHDLCStatus & DRxFD ;
        }
        pRxBDPtr = HDMARXPTRA ;
        gFrameCount[HDLC_A]++;
    }

    if(IntHDLCStatus & (RxMOV|DRxNL|DRxNO)){
        if(IntHDLCStatus & RxMOV) {
            gHRxStatus[HDLC_A].DMARxMOV++ ;
            HSTATATA = IntHDLCStatus & RxMOV ;
            Print("\r HDLCA Rx Fail");
            return ;
        }
        if(IntHDLCStatus & DRxNL) {
            gHRxStatus[HDLC_A].DMARxNL++ ;
            HSTATATA = IntHDLCStatus & DRxNL ;
            Print("\r HDLCA Rx Fail");
            return ;
        }
        if(IntHDLCStatus & DRxNO) {
            gHRxStatus[HDLC_A].DMARxNO++ ;
            HSTATATA = IntHDLCStatus & RxNO ;
            Print("\r HDLCA Rx Fail");
            return ;
        }
    }

    if (!CheckRxStatusHDLCA(HDLC_A, 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**.

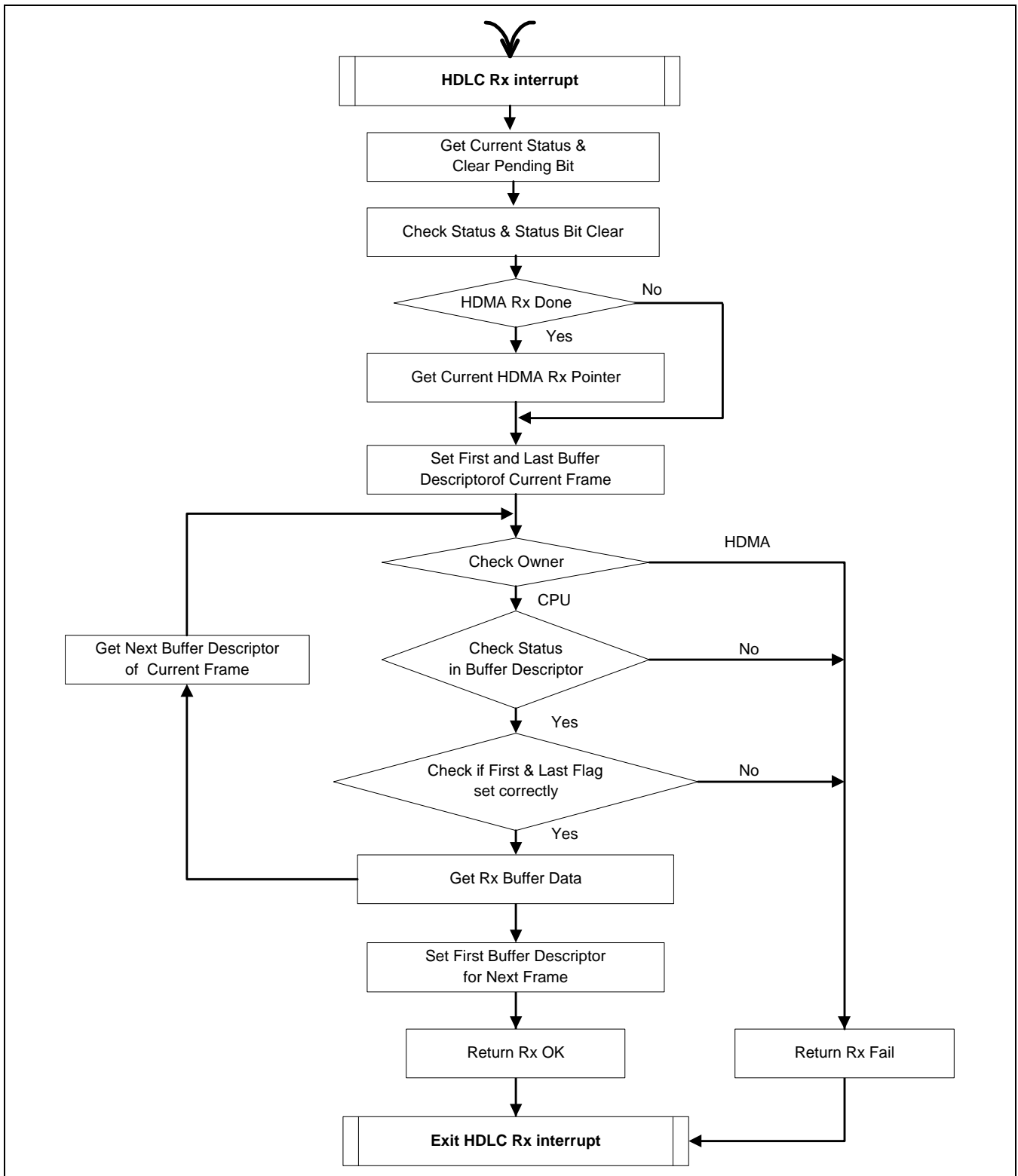


Figure 3-30. HDLC Frame reception flow

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.

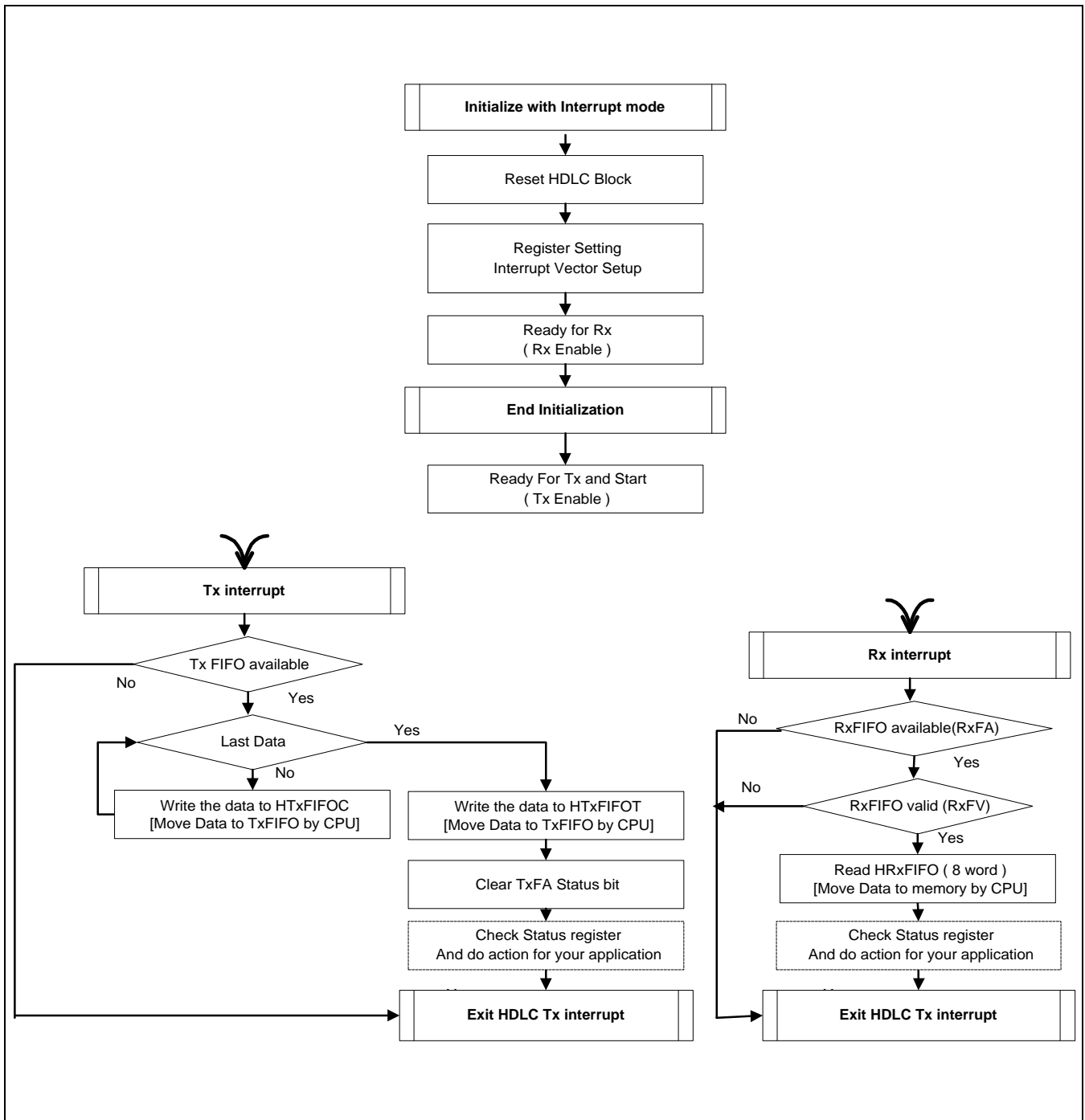


Figure 3-31. Interrupt mode operation flow

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 & 0xFFFFF000 ;
            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() ;
}
```

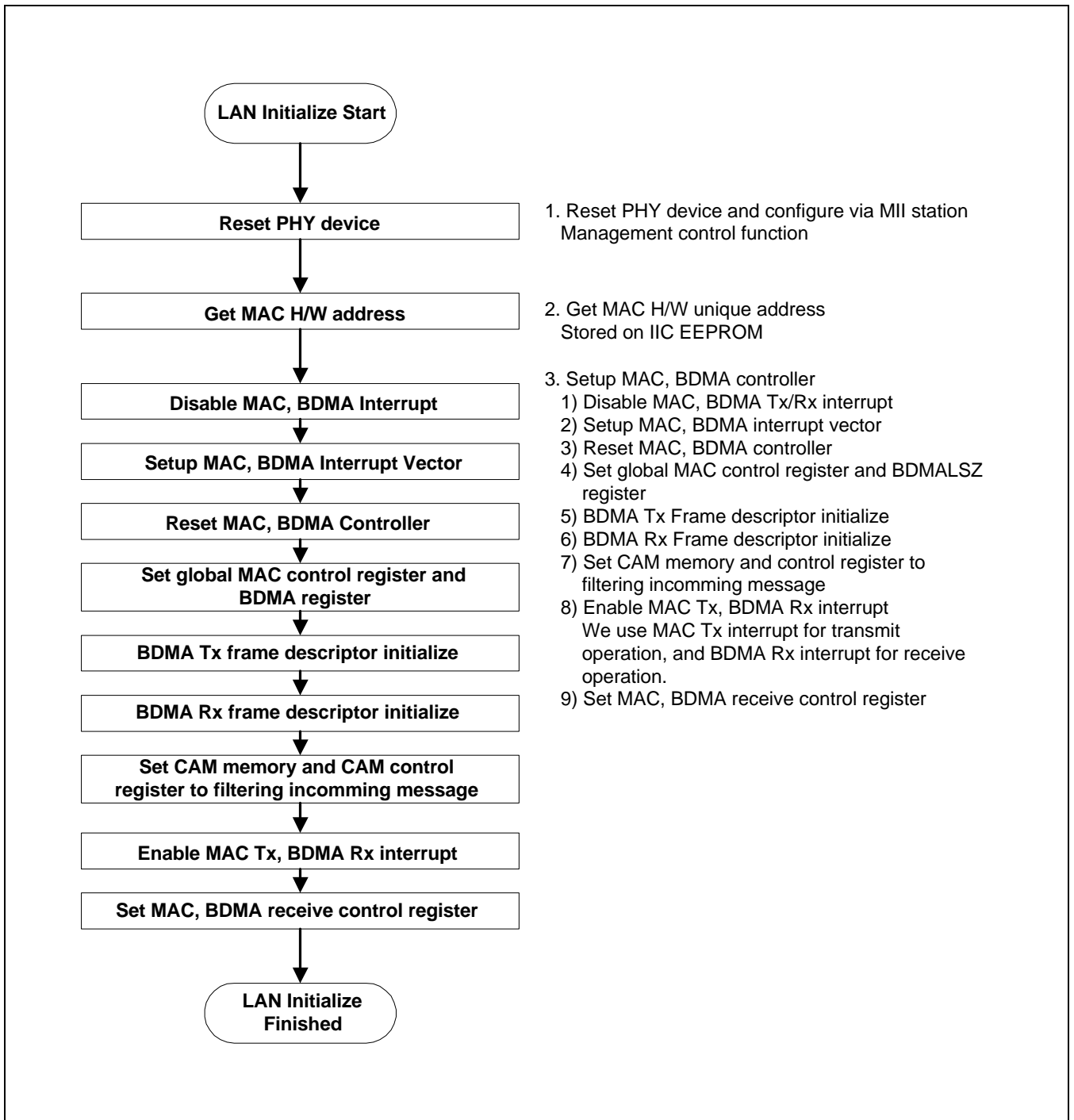


Figure 3-32. LAN Initialize flow

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-negotiation
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 ;

#ifdef 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;
```

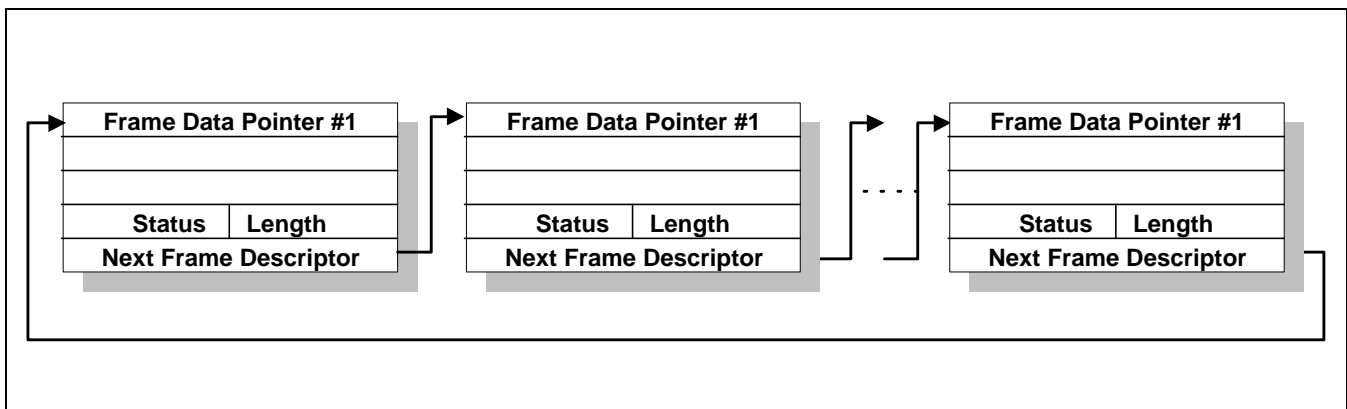


Figure 3-33. BDMA Frame Descriptor Structure

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 owneded 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 owneded 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. Change 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.TxDeferredErr++ ;
    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**.

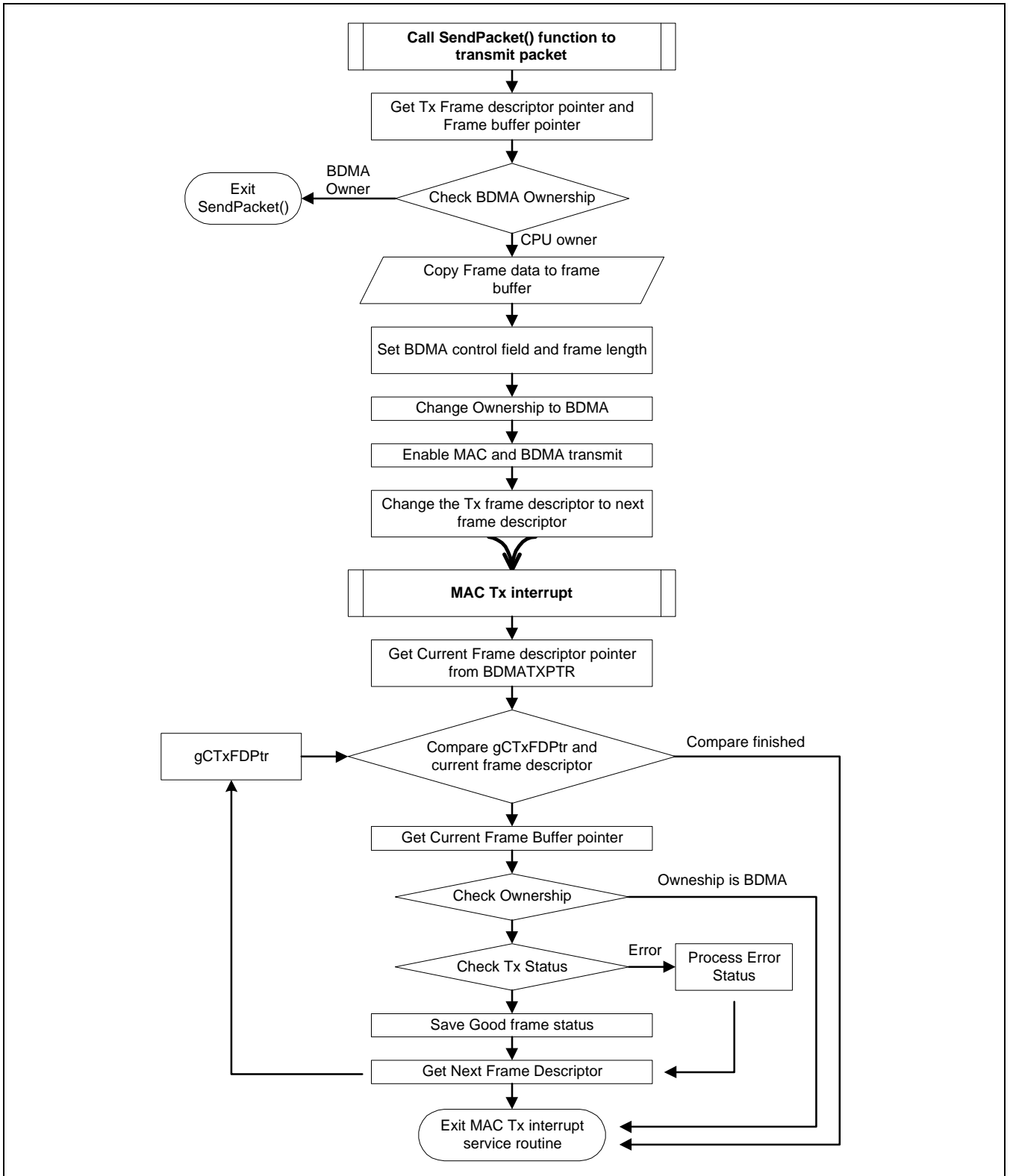


Figure 3-34. Ethernet Frame transmit flow

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 BDMASAT 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 BDMASAT 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
 * Ethernet Frame is received in BDMA_Rx_isr
 */
void BDMA_Rx_isr(void)
{
    sFrameDescriptor *pRxFDptr ;
    U32 RxStatus, FrameLength ;
    U32 CRxPtr;
    U32 sBdmaStat ;
    U8 *pFrameData ;
#ifdef KS32C5000_BUG_FETCH
    U32 FstData,FstDataSave,*FstFrameData ;
#endif

    // Step 1. Get current frame descriptor and status
    CRxPtr = BDMARXPTR ;
    sBdmaStat = BDMASAT ;

    // Step 2. Clear BDMA status register bit by write 1
    BDMASAT |= S_BRxRDF ;
    gsBdmaRxStatus.BdmaRxCnt++ ;

    do {
        // Step 3. Check Null List Interrupt
        if ( BDMASAT & S_BRxNL ) {
            BDMASAT |= S_BRxNL ;
            gsBdmaRxStatus.BRxNLErr++ ;
            MacInitialize() ;
            break ;
        }

        // Step 4. Get Rx Frame Descriptor
        pRxFDptr = (sFrameDescriptor *)gCRxPtr ;
        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.AlignErr++ ;
        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 ) {

```



```
        BDMASAT |= 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**.

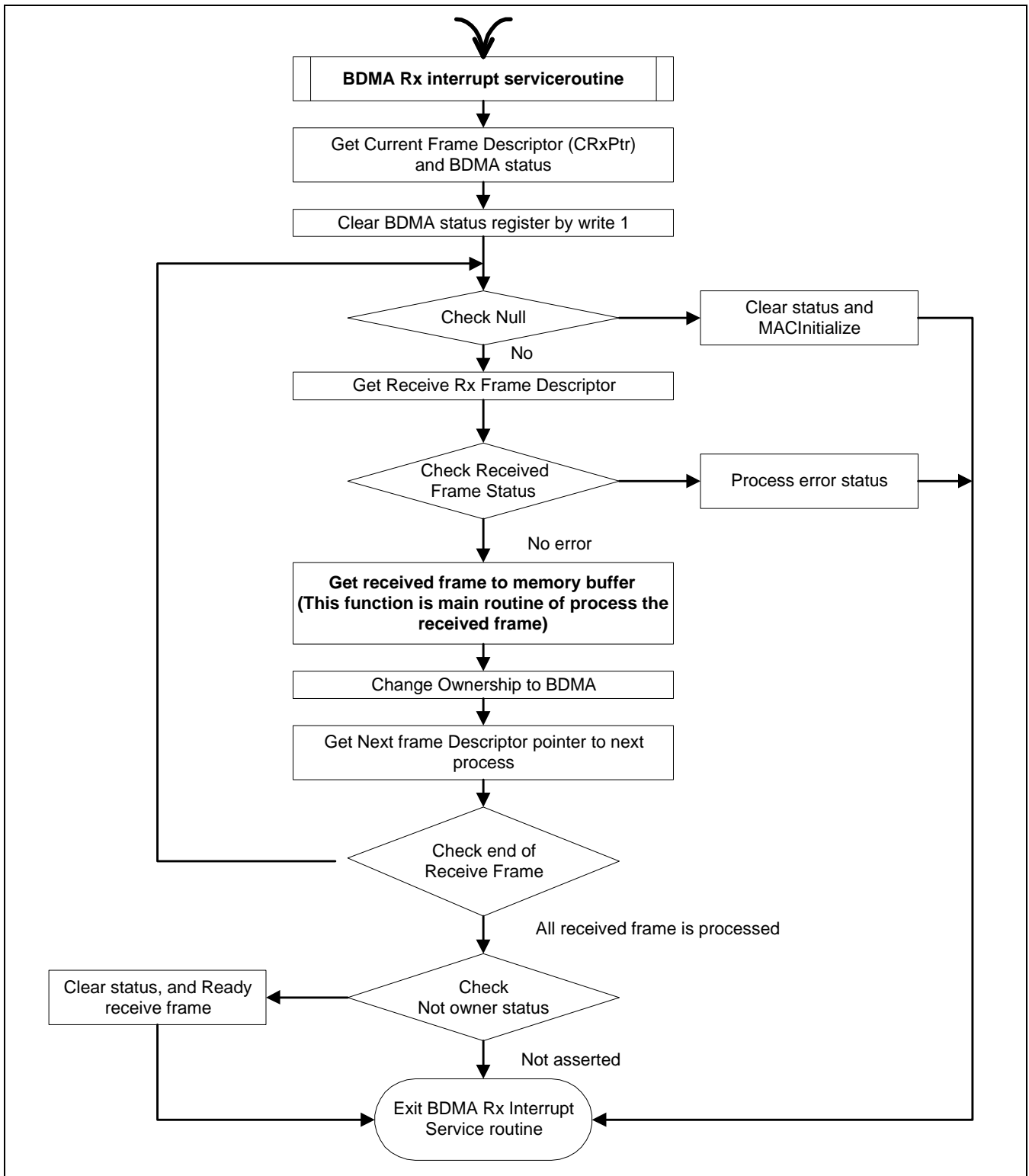


Figure 3-35. Ethernet Frame Reception Flow

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