The bootstrap procedure of u-boot.
Take NDS32 architecture as an example.

Macpaul Lin

macpaul <at> gmail.com

2012/Jan/10

Outline

- Introduction.
- How to pack a boot loader?
- Overview – The bootstrap procedure
- Binary relocation and memory remap.
- Vector setup.
- Low level initialization.
- Stack setup and jump into C functions.
- Early board and peripherals initialization.
- General Relocation.
- Common board and other peripherals initialization.
- Summary
- Appendix

Introduction

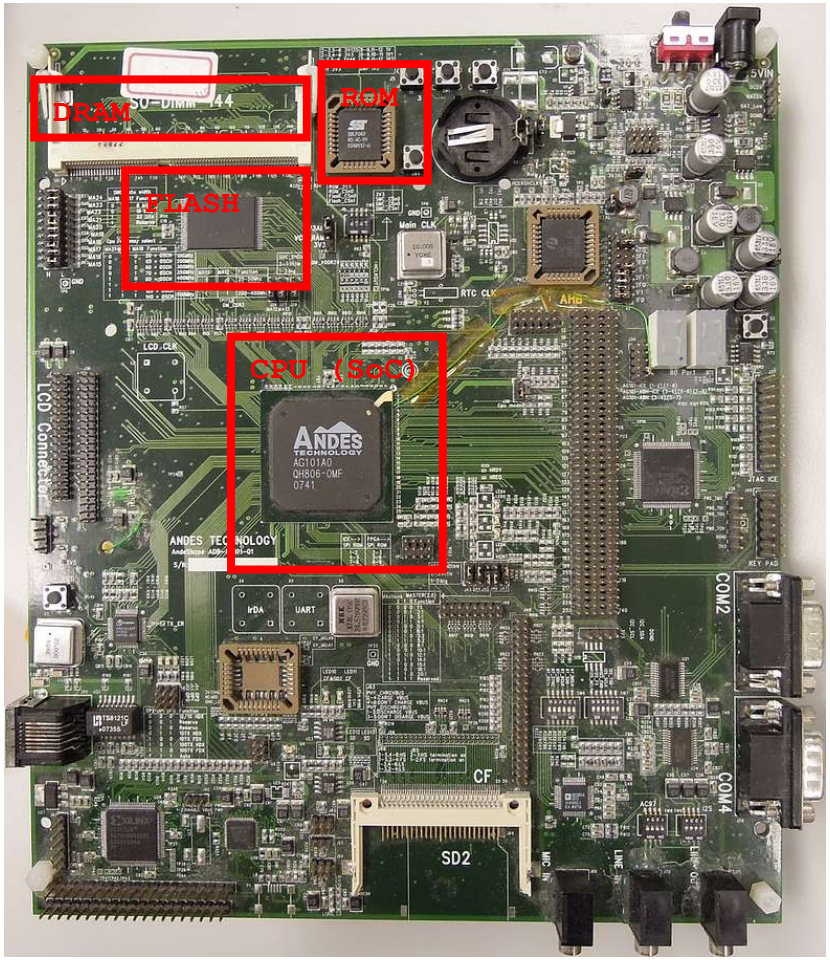
- This slide only discuss the bootstrap procedure of u-boot.
  - We will also discuss device drivers which will be touched only in early initialization stages.
  - The license of this slide
    - CC-SA: http://creativecommons.org/licenses/by-sa/3.0/tw/
- We will take N1213 CPU core which is NDS32 RISC architecture as the reference.
- U-boot is a boot loader for embedded system.
  - Supports PPC, ARM, AVR32, MIPS, x86, m68k, nios, microblaze, blackfin, and NDS32.
  - Wikipedia: http://en.wikipedia.org/wiki/Das_U-Boot
  - Project: http://www.denx.de/wiki/U-Boot/WebHome
  - Git: git://git.denx.de/u-boot.git
  - This slides has been wrote based on the version
    - ftp://ftp.denx.de/pub/u-boot/u-boot-2011.12.tar.bz2
  - The bootstrap procedure will be improved continuously, so please follow up the most recent develop discussion on mailing list.
    - Mailing list: http://lists.denx.de/mailman/listinfo/u-boot

Introduction

- What is boot loader?
  - http://en.wikipedia.org/wiki/Boot_loader#Boot_loader
  - Functionality
    - The main task of boot loader is to prepares CPU and RAM to access the nonvolatile devices to load OS into ram.
    - It provides firmware upgrade and fail-safe functions.
    - It could run basic diagnostic and testing.
    - You can develop simple application on u-boot.
      - You can even use u-boot with non-OS frame buffer display (VFD) on some devices..
  - But the challenge of a boot loader is how to boot itself into the memory appropriately.
    - This might be the most important task of a boot loader.

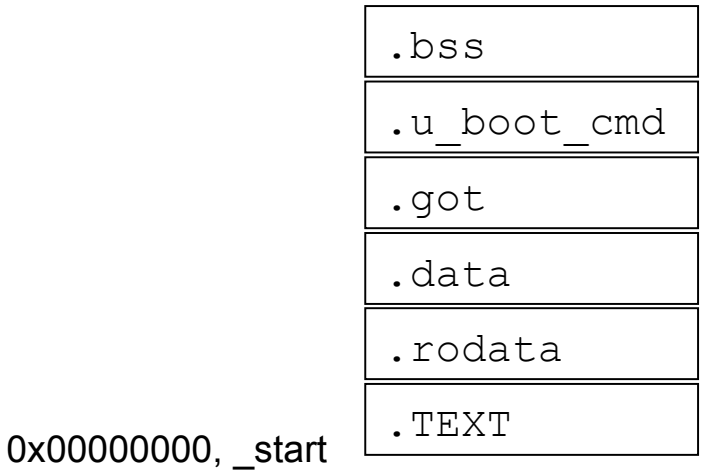
Introduction

- Bootstrap procedure.
  - Purpose
    - The boot loader is usually stored in FLASH or EEPROM.
    - The boot loader must do the very beginning setup and then copy itself from FLASH or EEPROM into DRAM then continue setting hardware.
  - How
    - After power on, CPU will load the first line of machine code (the beginning of a boot loader) from FLASH or EEPROM.
      - The $PC (program counter register) is usually the initial value (0x0) which points to the base address of FLASH or ROM.
    - Then boot loader will setup DRAM controller and copy itself into DRAM.
  - The program of bootstrap procedure is usually called the “boot code” or “startup code”.



How to pack a boot loader?

- How to pack a boot loader into FLASH or ROM?
  - The linker script will pack the binary of all the above procedures of a boot loader and other functions in to a single binary file.
  - This binary file is the boot loader which will be burned into FLASH or EEPROM.
  - The base address of .TEXT could be changed depends on the initial value of the $PC which will vary for different SoC.

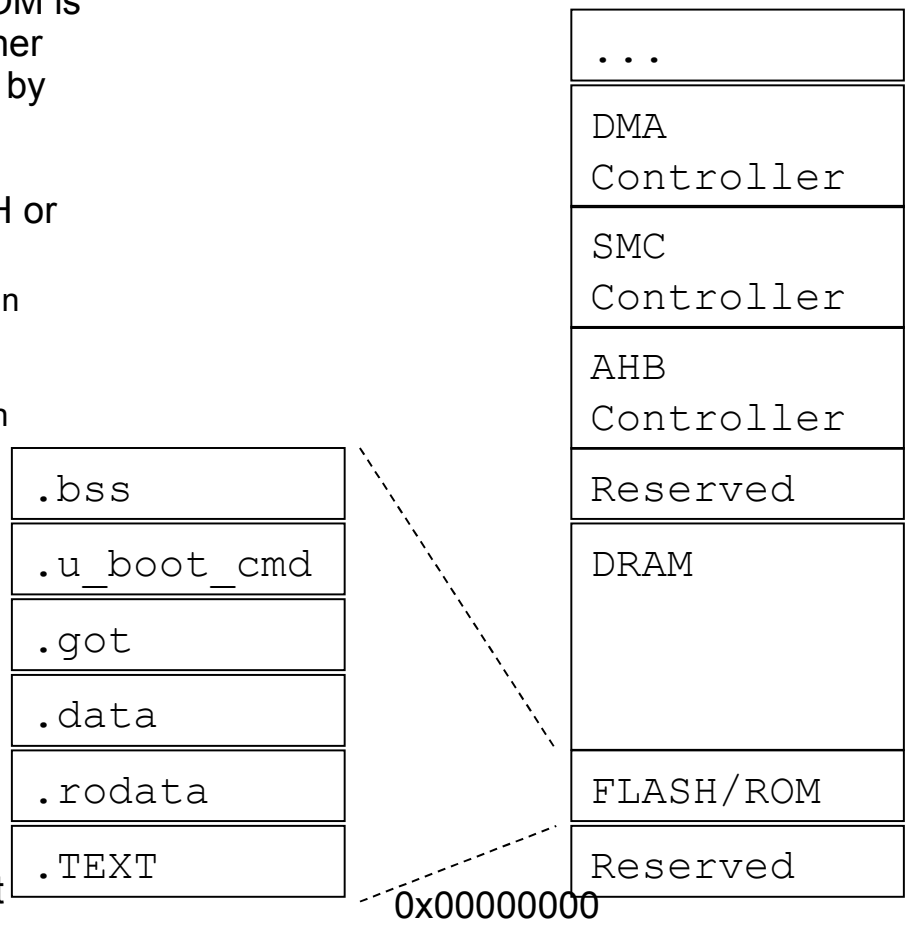



```
arch/nds32/cpu/n1213/u-boot.lds:
OUTPUT_FORMAT("elf32-nds32", "elf32-nds32", "elf32-nds32")
OUTPUT_ARCH(nds32)
ENTRY(_start)
SECTIONS
{
        . = ALIGN(4);
        .text :
        {
                arch/nds32/cpu/n1213/start.o    (.text)
                *(.text)
        }

        . = ALIGN(4);
        .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }

        . = ALIGN(4);
        .data : { *(.data*) }

        . = ALIGN(4);

        .got : {
                __got_start = .;
                 *(.got.plt) *(.got)
                __got_end = .;
        }

        . = .;
        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;

        . = ALIGN(4);

        _end = .;

        .bss : {
                __bss_start = .;
                *(.bss)
                 . = ALIGN(4);
                __bss_end__ = .;
        }
}
```

How to pack a boot loader?

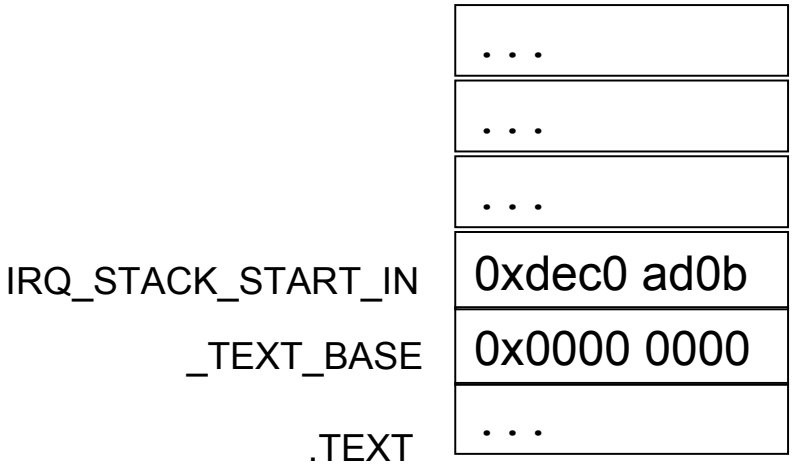
- If the initial value of a $PC is not equal to zero (the base address of FLASH or ROM is not at 0x0), the starting address and other linking address in u-boot can be shifted by an offset value "CONFIG_SYS_TEXT_BASE" to be execute correctly when it runs in FLASH or in ROM.
  - CONFIG_SYS_TEXT_BASE is defined in configuration file "include/configs/adp-ag101.h"
  - start.S will use this value to do relocation calculation.

```
arch/nds32/cpu/n1213/start.S:

/* Note: TEXT_BASE is defined by the (board-
     dependent) linker script */
.globl _TEXT_BASE
_TEXT_BASE:
        .word   CONFIG_SYS_TEXT_BASE
```



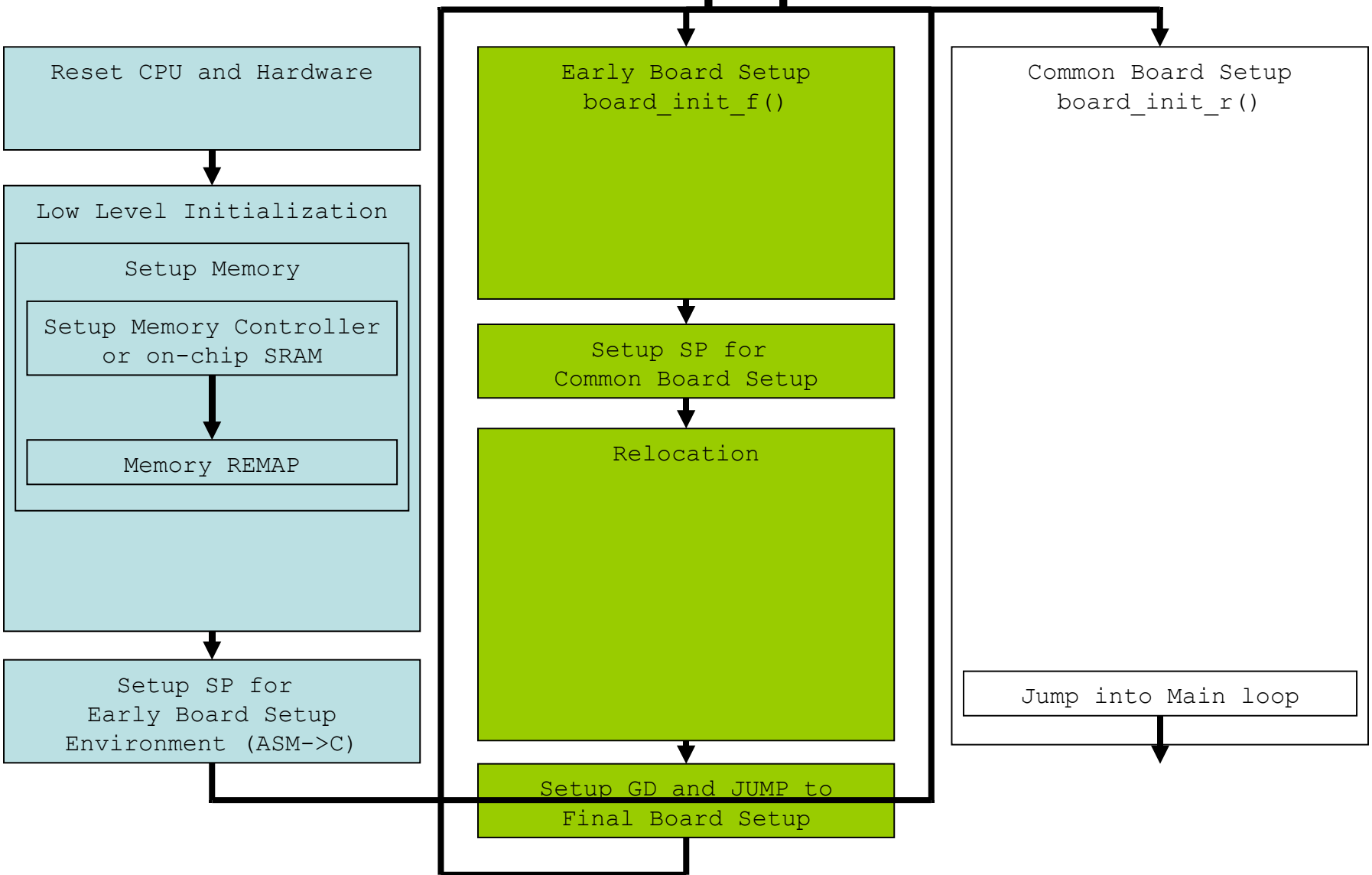
How to pack a boot loader?

- When u-boot copies itself into RAM, it will use some symbols to calculate the offset address for relocation and memory layout setup.
  - These symbols will be inserted into the ELF and linker will convert them into values to calculate relative address.
  - Some magic numbers and offsets will also be inserted into binary for calculating offsets in runtime.
  - NDS32 current won't separate IRQ and FIQ stack.




```
arch/nds32/cpu/n1213/start.S:
/*
 * These are defined in the board-specific linker
    script.
 * Subtracting _start from them lets the linker
    put their
 * relative position in the executable instead of
    leaving
 * them null.
 */
#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
        .word 0x0badc0de

/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
        .word 0x0badc0de
#endif

/* IRQ stack memory (calculated at run-time) + 8
    bytes */
.globl IRQ_STACK_START_IN
IRQ_STACK_START_IN:
        .word 0x0badc0de
```

Overview – The bootstrap procedure

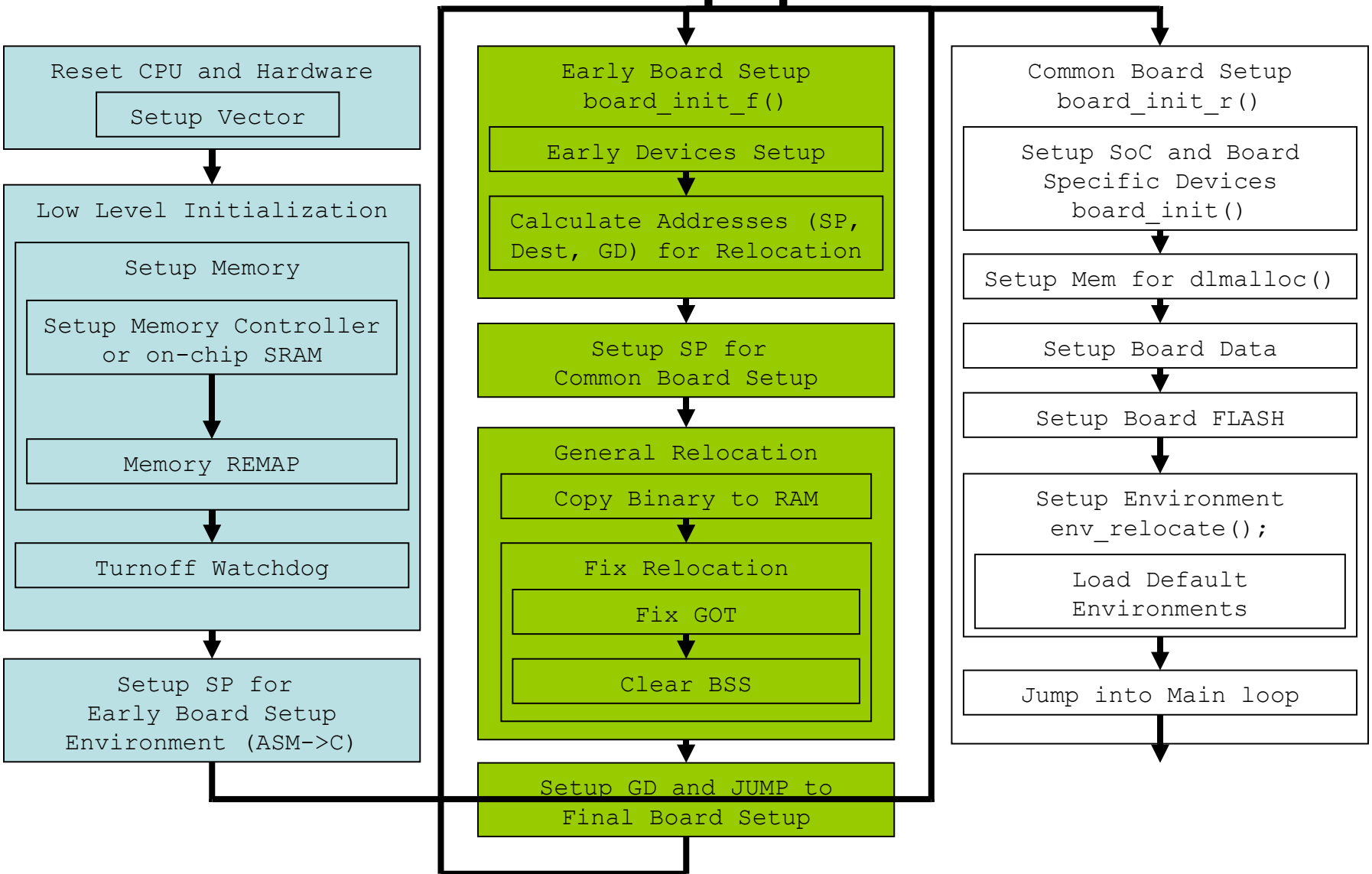


Overview – The bootstrap procedure

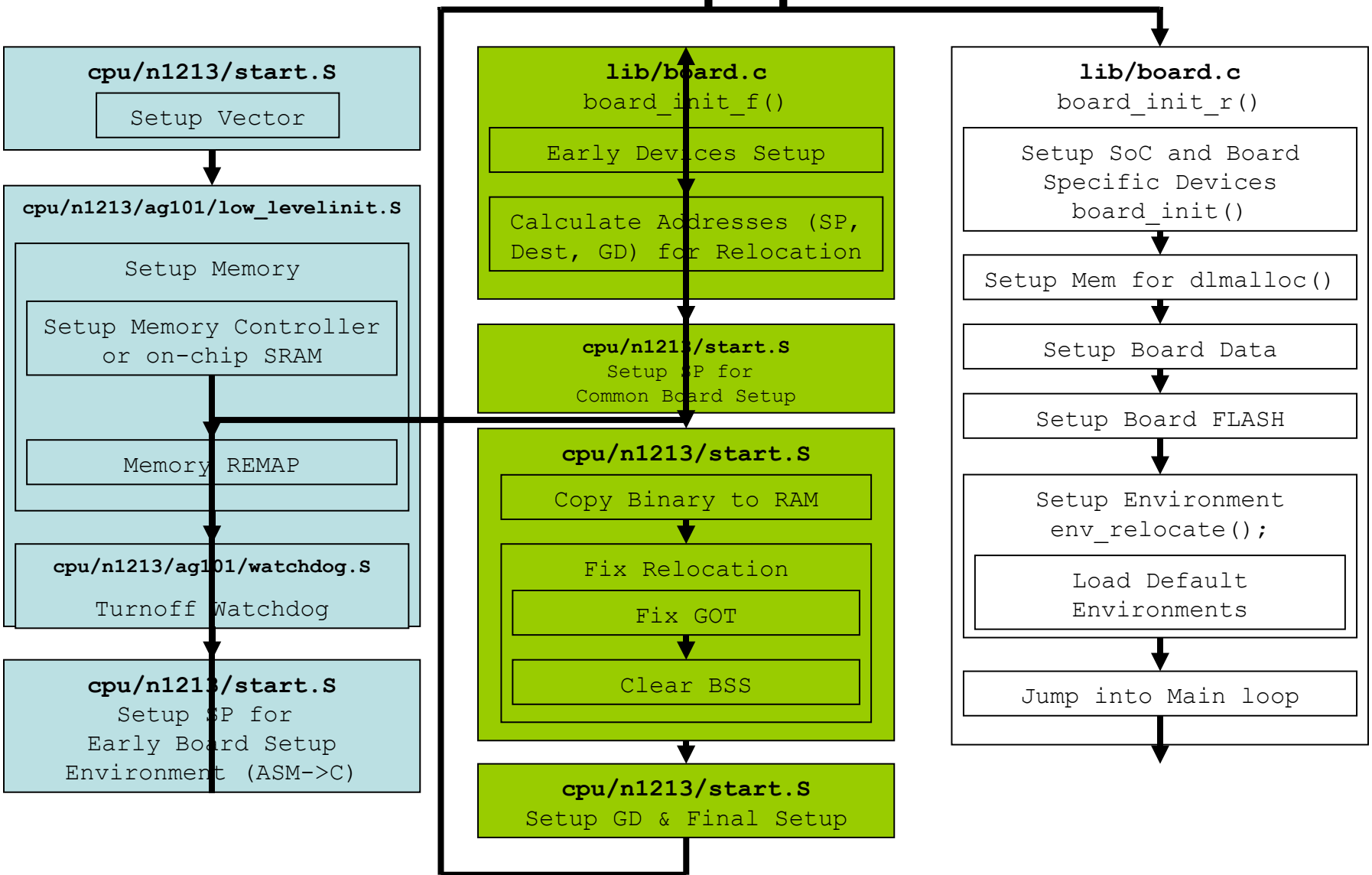
- Example
  - The description of boot code (startup code) in Start.S.

```
/*
 * Andesboot Startup Code (reset vector)
 *
 *      1.      bootstrap
 *              1.1 reset - start of u-boot
 *              1.2 to superuser mode - as is when reset
 *              1.3 Do lowlevel_init
 *                      - (this will jump out to lowlevel_init.S in SoC)
 *                      - (lowlevel_init)
 *              1.4 Do Memory Remap if it is necessary.
 *              1.5 Turn off watchdog timer
 *                      - (this will jump out to watchdog.S in SoC)
 *                      - (turnoff_watchdog)
 *      2.      Do critical init when reboot (not from mem)
 *      3.      Relocate andesboot to ram
 *      4.      Setup stack
 *      5.      Jump to second stage (board_init_r)
 */
```

Overview – The bootstrap procedure



Overview – The bootstrap procedure

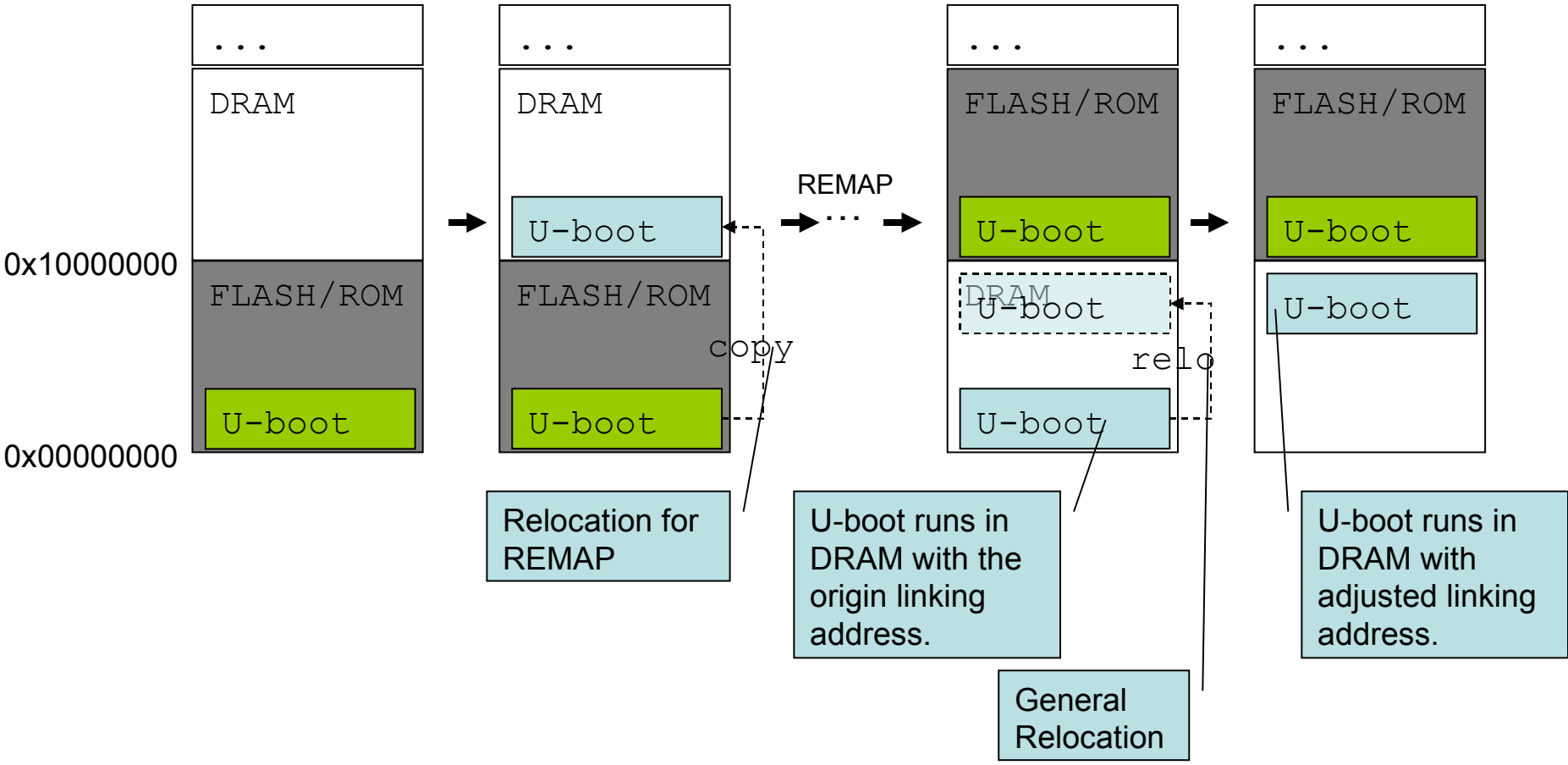


Binary relocation and memory remap.

- Binary relocation.
  - "Relocation is the process of assigning load addresses to various parts of [a] program and adjusting the code and data in the program to reflect the assigned addresses.“
    - John R. Levine (October 1999). "Chapter 1: Linking and Loading". Linkers and Loaders. Morgan-Kauffman. p. 5. ISBN 1-55860-496-0.
  - In other words, it means copy the binary from a source address to a new destination address but the binary must be able to execute correctly.
    - Doing memory relocation usually need to adjust the linking address offsets and global offset table through all binary.
    - If you don’t want to do address adjustment, you must copy the binary to the destination address which exactly “the same as” its origin address.
      - This requires the mechanism called memory “remap”.
  - Two relocation will happened in u-boot.
    - One is relocation with remap, which doesn’t need to do address adjustment.
    - The other is the general relocation (or, generic relocation), which need to do address adjustment based on dynamic calculated offsets relative to the top of the DRAM.

Binary relocation and memory remap.

- Binary relocation.

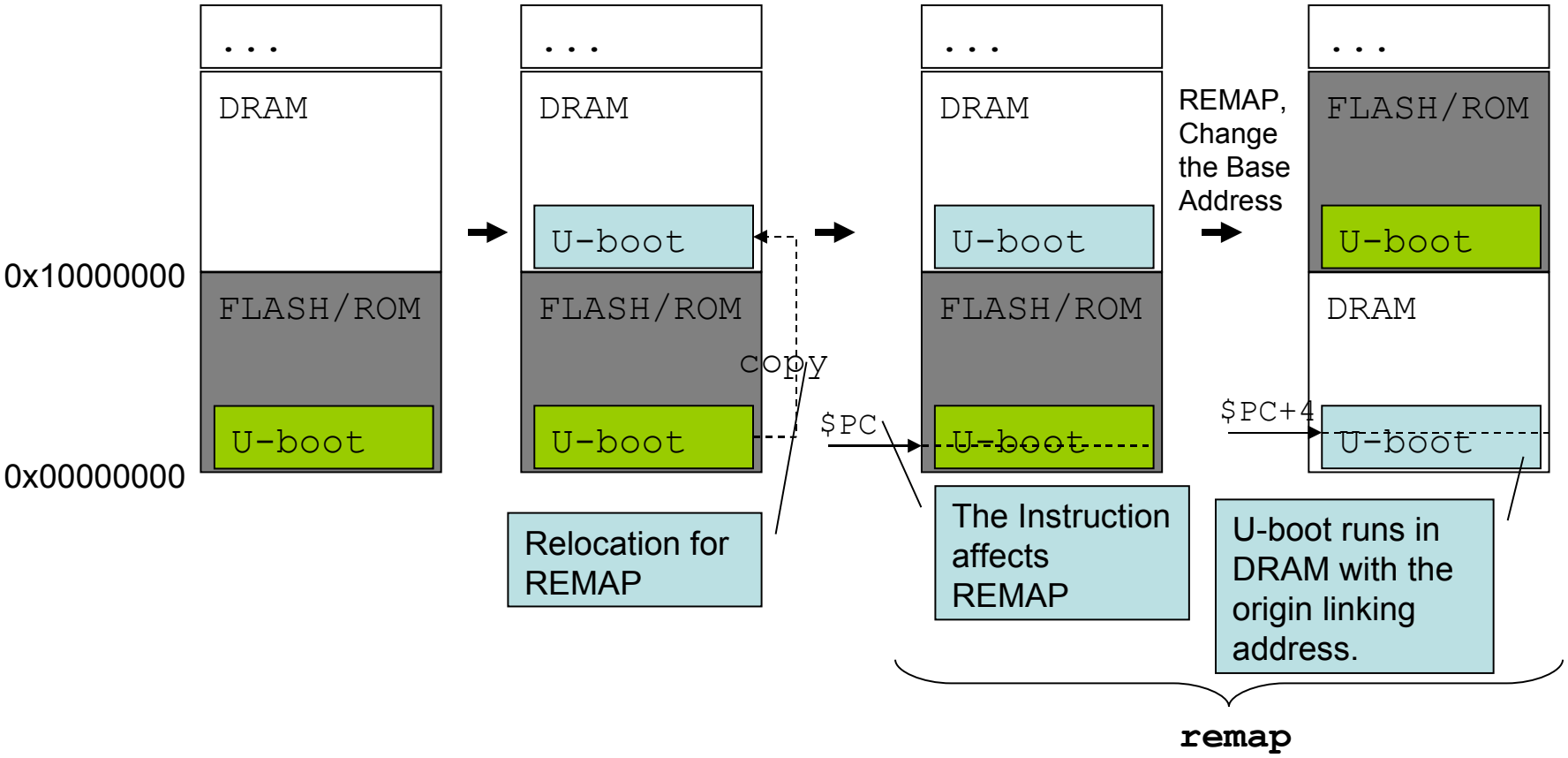


Binary relocation and memory remap.

- Memory remap.
  - Boot loader must copy itself into RAM even it ran in ROM at the very beginning.
    - Variable cannot be modified when it is stored in ROM.
      - Boot loader must copy itself to RAM to update the variables correctly.
      - Function calls relies on stack operation which is also should be stored in RAM.
    - For example, in some CPU architecture or system, ROM is assigned at with base address 0x00000000, where is the initialize value of $PC.
      - Boot loader is usually linked at the address same as the initialize value of $PC.
      - Base address exchange between ROM and RAM is usually happened when doing remapping.
      - You will need to do memory remap after boot loader has copied itself into RAM to make sure the execution correctness.

Memory relocation and remap.

- Memory remap.



Binary relocation and memory remap.

- Memory remap.
  - Not all architecture or SoC need to do memory remap, it is optional.
    - Some architecture provides a simple start-up code with remap runs before boot loader, like ARM.
    - The base address of ROM on some SoC is not begin at 0x00000000.
      - Which also means the initial value of $PC of this SoC could be configured with other value.
    - Some architecture or SoC doesn't allocate vector table at 0x00000000.
      - The base address of DRAM may not required to be 0x00000000.
      - It's configurable if the CPU has a system register to configure the base address of vector table.

Binary relocation and memory remap.

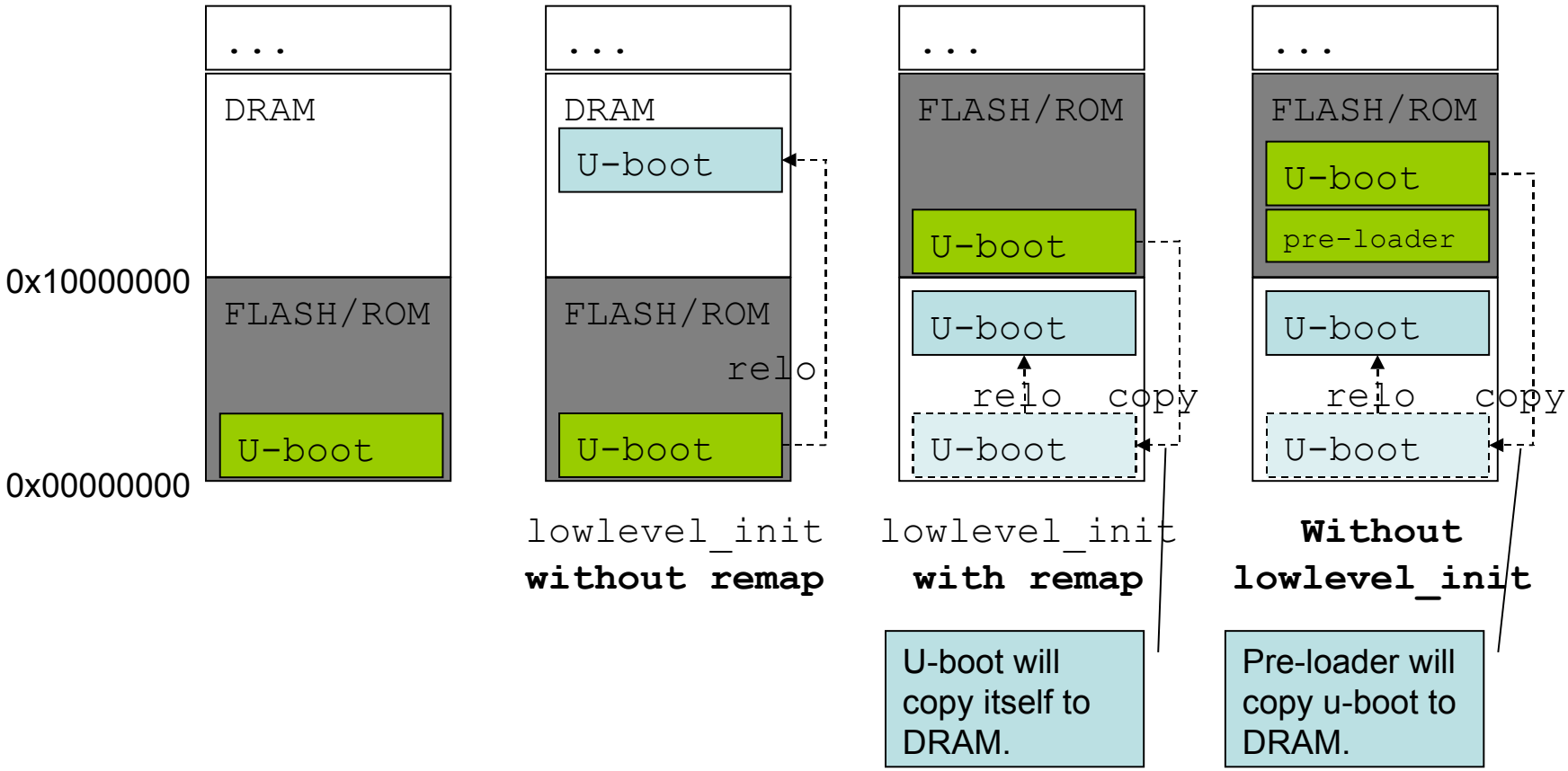
- Memory remap is optional, it depends on the configuration of your system.
  - Most architecture doesn't require to run Linux with storing vector table at 0x00000000.
  - Memory remap is usually done with-in low level initialization.
    - You can also do low level initialization without memory remap which is configurable.
- General binary relocation is a must for current implementation of u-boot.
  - U-boot will calculate the reserved space from the top of the RAM for stack and other purpose.
  - Finally u-boot will relocate itself to the address below this reserved area, which is near the top of the RAM.

Binary relocation and memory remap.

- Low level initialization.
  - Low level initialization (lowlevel_init) will perform memory setup configuration and remap.
    - Memory remap is optional.
  - If another boot loader (pre-loader) runs before u-boot, the boot loader must done lowlevel_init for u-boot.
    - Hence u-boot should be configured with “CONFIG_SKIP_LOWLEVEL_INIT”.
      - U-boot won’t perform lowlevel_init and remap will be skipped.
    - This pre-loader will copy u-boot without address adjustment from FLASH or ROM into DRAM.
      - U-boot will perform general relocation itself.
      - The base link address CONFIG_SYS_TEXT_BASE should be the same as the destination which pre-loader will put u-boot.

Binary relocation and memory remap.

- You could choose one of the 3 types of lowlevel_init and remap combination when u-boot is doing bootstrap.



Vector setup.

- Vector table is allocated at the beginning of the u-boot binary.
  - After general relocation u-boot will usually copy the vector table to the correct address and adjust the address where the real vectors exist.
- When CPU loads u-boot from FLASH or ROM, it will jump to the first vector to do “reset”.
  - In reset vector, CPU will do CPU core initialization and check if lowlevel_init is required.

arch/nds32/cpu/n1213/start.S

```
.globl _start

_start: j       reset
        j       tlb_fill
        j       tlb_not_present
        j       tlb_misc
        j       tlb_vlpt_miss
        j       machine_error
        j       debug
        j       general_exception
        j       syscall
        j       internal_interrupt  ! H0I
        j       internal_interrupt  ! H1I
        j       internal_interrupt  ! H2I
        j       internal_interrupt  ! H3I
        j       internal_interrupt  ! H4I
        j       internal_interrupt  ! H5I
        j       software_interrupt  ! S0I
        .balign 16
```

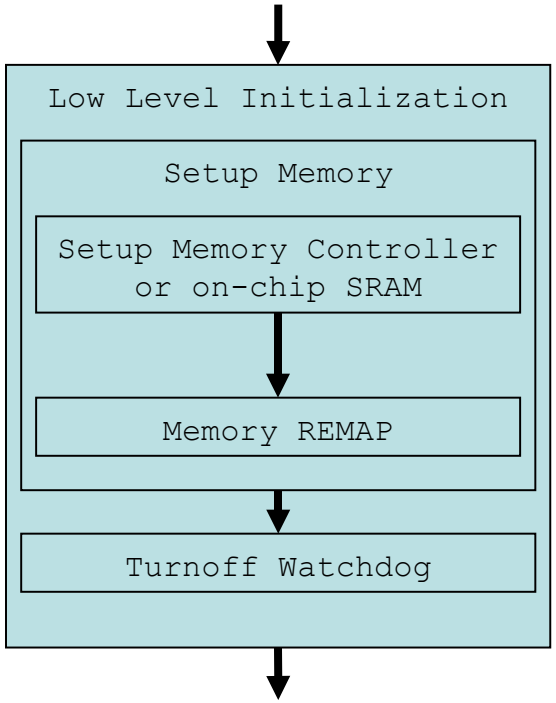
Vector setup.

- reset:
  - Configure $MISC_CTL and $IVB to make sure the vectors are 4 bytes aligned at the address 0x00000000.
- load_lli:
  - This will jump to the procedure lowlevel_init and then return.
- turnoff_wtdog:
  - This will jump to the procedure load_turnoff_watchdog and then return.

```
arch/nds32/cpu/n1213/start.S
/*
 * The bootstrap code of nds32 core
 */
reset:
set_ivb:
        li      $r0, 0x0

        /* turn on BTB */
        mtsr    $r0, $misc_ctl
        /* set IVIC, vector size: 4 bytes, base:
    0x0 */
        mtsr    $r0, $ivb

load_lli:
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
        jal     load_lowlevel_init
        jral    $p0
#endif

/*
 * Set the N1213 (Whitiger) core to superuser mode
 * According to spec, it is already when reset
 */
turnoff_wtdog:
#ifndef CONFIG_SKIP_TRUNOFF_WATCHDOG
        jal     load_turnoff_watchdog
        jral    $p0
#endif
```

Low level initialization

- The purpose if lowlevel_init is to configure the power and memory setting by when RAM are not initialized.
- The main task of lowlevel_init is to setup memory controller and relatives to make sure the DRAM could be operate correctly.
  - Setup timing and clock of DRAM.
  - Setup power related issue of DRAM.
  - Setup base address of DRAM.
  - Perform memory remap of u-boot.
- To turn off watchdog timer is to avoid hardware reset triggered by random value inside an enabled watchdog timer during bootstrap.



Low level initialization

- ASM macro load_lowlevel_init
  - will jump to the real lowlevel_init macro and return.
- ASM macro load_turnoff_watchdog
  - will jump to the real load_turnoff_watchdog macro and return.

```
arch/nds32/cpu/n1213/start.S
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
load_lowlevel_init:
        la  $r6, lowlevel_init
        la  $r7, load_lli + 4
        sub $p0, $r6, $r7
        add $p0, $p0, $lp
ret
#endif

#ifndef CONFIG_SKIP_TRUNOFF_WATCHDOG
load_turnoff_watchdog:
        la  $r6, turnoff_watchdog
        la  $r7, turnoff_wtdog + 4
        sub $p0, $r6, $r7
        add $p0, $p0, $lp
ret
#endif
```

Low level initialization

- There are 2 main task inside lowlevel_init
  - Jump to mem_init
    - Configuration DRAM controller and setup base address
  - Jump to remap
    - Memory remap.

arch/nds32/cpu/n1213/ag101/lowlevel_init.S

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
.globl  lowlevel_init
lowlevel_init:
        move    $r10, $lp

        led     0x0
        jal     mem_init

        led     0x10
        jal     remap

        led     0x20
        ret     $r10
```

Low level initialization

- mem_init
  - Setup static memory controller (SMC).
    - Setting control and timing registers.
    - To make sure the timing of reading from FLASH or ROM is correct.
  - Setup AHB controller
    - Setting base address of DRAM.
  - Setup power management controller (PMU).
    - Setting the required power parameters for DRAM.

arch/nds32/cpu/n1213/ag101/lowlevel_init.S

```
mem_init:
        move    $r11, $lp

        /*
         * mem_init:
         *      There are 2 bank connected to FTSMC020
    on AG101
         *      BANK0: FLASH/ROM (SW5, J16), BANK1:
    OnBoard SDRAM.
         *      we need to set onboard SDRAM before
    remap and relocation.
         */
        led     0x01
        write32 SMC_BANK0_CR_A, SMC_BANK0_CR_D
            ! 0x10000052
        write32 SMC_BANK0_TPR_A, SMC_BANK0_TPR_D
            ! 0x00151151

        /*
         * config AHB Controller
         */
        led     0x02
        write32 AHBC_BSR6_A, AHBC_BSR6_D

        /*
         * config PMU controller
         */
        /* ftpmu010_dlldis_disable, must do it in
    lowleve_init */
        led     0x03
        setbf32 PMU_PDLLCR0_A, FTPMU010_PDLLCR0_DLLDIS
            ! 0x00010000
```

Low level initialization

- mem_init (cont.)
  - Setup SDRAM controller (SDMC).
    - Setting control and timing registers.
    - Setting other required parameters.

arch/nds32/cpu/n1213/ag101/lowlevel_init.S

```
        /*
         * config SDRAM controller
         */
        led     0x04
        write32 SDMC_TP1_A, SDMC_TP1_D
            ! 0x0001I312
        led     0x05
        write32 SDMC_TP2_A, SDMC_TP2_D
            ! 0x00480180
        led     0x06
        write32 SDMC_CR1_A, SDMC_CR1_D
            ! 0x00002326

        led     0x07
        write32 SDMC_CR2_A, FTSDMC021_CR2_IPREC
            ! 0x00000010
        wait_sdram

        led     0x08
        write32 SDMC_CR2_A, FTSDMC021_CR2_ISMR
            ! 0x00000004
        wait_sdram

        led     0x09
        write32 SDMC_CR2_A, FTSDMC021_CR2_IREF
            ! 0x00000008
        wait_sdram

        led     0x0a
        move    $lp, $r11
        ret
```

Low level initialization

- remap:
  - For CPU support only NDS32 V0 ISA
    - There is no “mfusr” instruction for you to access $PC directly.
    - The work around is to use branch and then access $LP which will be the same as $PC.
  - Remapping
    - First configure the base address of DRAM in SDRAM controller.

arch/nds32/cpu/n1213/ag101/lowlevel_init.S

```
remap:
        move    $r11, $lp
#ifdef __NDS32_N1213_43U1H__    /* NDS32 V0 ISA - AG101
     Only */
        bal     2f
relo_base:
        move    $r0, $lp
#else
relo_base:
        mfusr   $r0, $pc
#endif /* __NDS32_N1213_43U1H__ */

        /*
         * Remapping
         */
        led     0x1a
        write32 SDMC_B0_BSR_A,
     SDMC_B0_BSR_D            ! 0x00001100

        /* clear empty BSR registers */
        led     0x1b
        li      $r4, CONFIG_FTSDMC021_BASE
        li      $r5, 0x0
        swi     $r5, [$r4 + FTSDMC021_BANK1_BSR]
        swi     $r5, [$r4 + FTSDMC021_BANK2_BSR]
        swi     $r5, [$r4 + FTSDMC021_BANK3_BSR]
```

Low level initialization

- remap:
  - CONFIG_MEM_REMAP
    - Setup the origin base address for SDRAM in $r4.
    - Setup the destination address for binary copy from ROM to RAM.
  - Set remap bit
    - Set REMAP bit into AHB Controller
      - After setting the REMAP bit, the base address of SDRAM and ROM will be exchanged.

arch/nds32/cpu/n1213/ag101/lowlevel_init.S

```
#ifdef CONFIG_MEM_REMAP
 led     0x11
        li      $r4, PHYS_SDRAM_0_AT_INIT               /*
      0x10000000 */
        li      $r5, 0x0
        la      $r1, relo_base                          /* get
      $pc or $lp */
        sub     $r2, $r0, $r1
        sethi   $r6, hi20(_end)
        ori     $r6, $r6, lo12(_end)
        add     $r6, $r6, $r2
1:
        lwi.p   $r7, [$r5], #4
        swi.p   $r7, [$r4], #4
        blt     $r5, $r6, 1b

        /* set remap bit */
        /*
         * MEM remap bit is operational
         * - use it to map writeable memory at 0x00000000, in
      place of flash
 * - before remap: flash/rom 0x00000000, sdram:
      0x10000000-0x4fffffff
         * - after  remap: flash/rom 0x80000000, sdram:
      0x00000000
         */
        led     0x1c
        setbf15 AHBC_CR_A, FTAHBC020S_CR_REMAP          ! 0x1

#endif /* #ifdef CONFIG_MEM_REMAP */
        move    $lp, $r11
2:
        ret
```

Low level initialization

- Note:
  - The mechanism inside AHBC when setting remap bit.
    - The following is quoted from the DATASHEET of FTAHBC020S.

- Remap function, switch base address of slave 4 and slave 6.
  - Activate remap function

    After applying the remap function, the new base address of slave 6 = the original base address of slave 4;

    the new base address of slave 4 = the original base address of slave 4 + space size of slave 6.
  - Note that the base address should be the boundary of the space size.

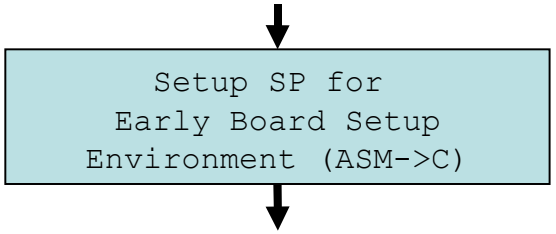
Low level initialization

- Turn off watchdog to avoid hardware reset accidently during bootstrap.
  - You can implement this in C language if the period of lowlevel_init is very short.

```
arch/nds32/cpu/n1213/ag101/watchdog.S

.text

#ifndef CONFIG_SKIP_TRUNOFF_WATCHDOG
ENTRY(turnoff_watchdog)

#define WD_CR           0xC
#define WD_ENABLE       0x1

        ! Turn off the watchdog, according to Faraday
     FTWDT010 spec
        li      $p0,
     (CONFIG_FTWDT010_BASE+WD_CR)       ! Get the addr
     of WD CR
        lwi     $p1, [$p0]          ! Get the config of WD
        andi    $p1, $p1, 0x1f      ! Wipe out useless bits
        li      $r0, ~WD_ENABLE
        and     $p1, $p1, $r0       ! Set WD disable
        sw      $p1, [$p0]          ! Write back to WD CR

        ! Disable Interrupts by clear GIE in $PSW reg
        setgie.d

        ret

ENDPROC(turnoff_watchdog)
#endif
```

Stack setup and jump into C functions.

- Stack setup is important before programs executes from ASM to C environment.
  - Compiler (gcc + bintuils) will have its own register usage and operation on stack to maintain the function calls.
  - It is also related to linking mechanism which implemented in compiler.
    - Different linking mode (PIC/non-PIC) will lead different usage among registers.
    - $sp, $lp, $gp, and $r15 must be set correctly for address offset calculation and function returns.
  - The linking method is quite different in PIC mode and in non-PIC mode.
    - The register layout and usage of $gp, $r15 will be different.
    - You must use PIC mode in nowadays u-boot.




```
nds32le-linux-objdump -D u-boot: (V1 ISA toolchain)
00000f18 <board_init_r>:
     f18:       3a 6f a1 bc     smw.adm $r6,[$sp],
     $r8,#0x6
     f1c:       45 d2 07 58     movi $gp,#132952
     f20:       42 ff 80 20     mfusr $r15,$pc
     f24:       41 d7 f4 00     add $gp,$r15,$gp
     f28:       51 ff ff fc     addi $sp,$sp,#-4
     f2c:       81 40           mov55 $r10,$r0
...
    1042:       dd 26           jral5 $r6
    1044:       44 fd f9 ca     movi $r15,#-132662
    1048:       40 f7 f4 00     add $r15,$r15,$gp
    104c:       dd 0f           jr5 $r15
    104e:       92 00           srli45 $r0,#0x0
```

Stack setup and jump into C functions.

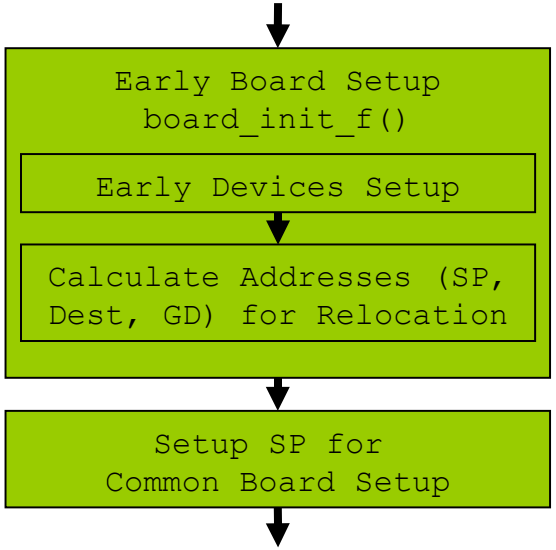
- In early stack setup, we just specify a compile-time parameter “CONFIG_SYS_INIT_SP_ADDR” to indicate where the early stack begins.
  - This depends on your memory layout design of your SoC.
  - It is configured in file “include/configs/adp-ag101.h”.
- __NDS32_N1213_43U1H__ implies the code is using NDS32 V0 ISA and cooresponding register layout.
- Then we jump to early board setup C function board_init_f().

arch/nds32/cpu/n1213/start.S

```
/*
 * Set stackpointer in internal RAM to call board_init_f
 * $sp must be 8-byte alignment for ABI compliance.
 */
call_board_init_f:
        li      $sp, CONFIG_SYS_INIT_SP_ADDR
        li      $r0, 0x00000000

#ifdef __PIC__
#ifdef __NDS32_N1213_43U1H__
/* __NDS32_N1213_43U1H__ implies NDS32 V0 ISA */
        la      $r15, board_init_f      ! store function
     address into $r15
#endif
#endif
        j       board_init_f            ! jump to
     board_init_f() in lib/board.c
```

Early board and peripherals initialization.

- After the memory is configured, early board setup (board_init_f) is the first C function which provides C level hardware setup capability by using the least amount of RAM.
  - You will do lot of stack and reserved memory calculation for general relocation.
  - You will also need to setup UART and console to help the debugging task.
  - Then you will need to pass the parameters to relocation_code ASM procedure to help on relocation and common board setup.



Early board and peripherals initialization.

- Setup the point of global data structure.
- Calculate the u-boot length between _start to __bss_end__.
- Initialize hardware peripherals with init_sequence[] which must be setup during early board setup.

arch/nds32/lib/board.c

```
void board_init_f(ulong bootflag)
{
        bd_t *bd;
        init_fnc_t **init_fnc_ptr;
        gd_t *id;
        ulong addr, addr_sp;

        /* Pointer is writable since we allocated a
     register for it */
        gd = (gd_t *) ((CONFIG_SYS_INIT_SP_ADDR) &
     ~0x07);

        memset((void *)gd, 0, GENERATED_GBL_DATA_SIZE);

        gd->mon_len = (unsigned int)(&__bss_end__) -
     (unsigned int)(&_start);

        for (init_fnc_ptr = init_sequence;
     *init_fnc_ptr; ++init_fnc_ptr) {
                if ((*init_fnc_ptr)() != 0)
                        hang();
        }
```

Early board and peripherals initialization.

- The content of init_sequence[].
  - Setup UART and console for debugging in early board setup and later.

arch/nds32/lib/board.c

```
init_fnc_t *init_sequence[] = {
#if defined(CONFIG_ARCH_CPU_INIT)
        arch_cpu_init,          /* basic arch cpu dependent
      setup */
#endif
#if defined(CONFIG_PMU) || defined(CONFIG_PCU)
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
        pmu_init,
#endif
#endif
        board_init,         /* basic board dependent setup */
#if defined(CONFIG_USE_IRQ)
        interrupt_init,     /* set up exceptions */
#endif
        timer_init,         /* initialize timer */
        env_init,           /* initialize environment */
        init_baudrate,      /* initialze baudrate settings */
        serial_init,        /* serial communications setup */
        console_init_f,     /* stage 1 init of console */
#if defined(CONFIG_DISPLAY_BOARDINFO)
        checkboard,         /* display board info */
#endif
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
        init_func_i2c,
#endif
        dram_init,          /* configure available RAM banks */
        display_dram_config,
        NULL,
};
```

Early board and peripherals initialization.

- Calculate the reserved memory areas and the stack address for general relocation.

arch/nds32/lib/board.c

```
 debug("monitor len: %08lX\n", gd->mon_len);
        /*
         * Ram is setup, size stored in gd !!
         */
        debug("ramsize: %08lX\n", gd->ram_size);

        addr = CONFIG_SYS_SDRAM_BASE + gd->ram_size;

#if !(defined(CONFIG_SYS_ICACHE_OFF) &&
     defined(CONFIG_SYS_DCACHE_OFF))
        /* reserve TLB table */
        addr -= (4096 * 4);

        /* round down to next 64 kB limit */
        addr &= ~(0x10000 - 1);

        gd->tlb_addr = addr;
        debug("TLB table at: %08lx\n", addr);
#endif

        /* round down to next 4 kB limit */
        addr &= ~(4096 - 1);
        debug("Top of RAM usable for U-Boot at:
     %08lx\n", addr);
```

Early board and peripherals initialization.

- Calculate the reserved memory areas and the stack address for general relocation. (cont.)
  - Reserve frame buffer here if you has LCD display or video devices.

arch/nds32/lib/board.c

```
#ifdef CONFIG_LCD
#ifdef CONFIG_FB_ADDR
        gd->fb_base = CONFIG_FB_ADDR;
#else
        /* reserve memory for LCD display (always full
     pages) */
        addr = lcd_setmem(addr);
        gd->fb_base = addr;
#endif /* CONFIG_FB_ADDR */
#endif /* CONFIG_LCD */

        /*
         * reserve memory for U-Boot code, data & bss
         * round down to next 4 kB limit
         */
        addr -= gd->mon_len;
        addr &= ~(4096 - 1);
```

Early board and peripherals initialization.

- Calculate the reserved memory areas and the stack address for general relocation. (cont.)
  - Reserve the memory area for dlmalloc memory management of malloc.
    - TOTAL_MALLOC_LEN
  - Setup board information structure.
  - Setup global data structure.

arch/nds32/lib/board.c

```
/*
 * reserve memory for malloc() arena
 */
addr_sp = addr - TOTAL_MALLOC_LEN;
debug("Reserving %dk for malloc() at: %08lx\n",
              TOTAL_MALLOC_LEN >> 10, addr_sp);
/*
 * (permanently) allocate a Board Info struct
 * and a permanent copy of the "global" data
 */
addr_sp -= GENERATED_BD_INFO_SIZE;
bd = (bd_t *) addr_sp;
gd->bd = bd;
memset((void *)bd, 0, GENERATED_BD_INFO_SIZE);
debug("Reserving %zu Bytes for Board Info at:
     %08lx\n",
      GENERATED_BD_INFO_SIZE, addr_sp);

addr_sp -= GENERATED_GBL_DATA_SIZE;
id = (gd_t *) addr_sp;
debug("Reserving %zu Bytes for Global Data at:
     %08lx\n",
       GENERATED_GBL_DATA_SIZE, addr_sp);
```

Early board and peripherals initialization.

- Calculate the reserved memory areas and the stack address for general relocation. (cont.)
  - If you need to specify IRQ stack, then reserve here.
    - Currently NDS32 doesn't distinguish FIQ and IRQ.
    - Adjustment stack and ABI compliance.

arch/nds32/lib/board.c

```
  /* setup stackpointer for exeptions */
        gd->irq_sp = addr_sp;
#ifdef CONFIG_USE_IRQ
        addr_sp -= (CONFIG_STACKSIZE_IRQ +
     CONFIG_STACKSIZE_FIQ);
        debug("Reserving %zu Bytes for IRQ stack at:
     %08lx\n",

     CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ,
     addr_sp);
#endif
        /* leave 3 words for abort-stack    */
        addr_sp -= 12;

        /* 8-byte alignment for ABI compliance */
        addr_sp &= ~0x07;
        debug("New Stack Pointer is: %08lx\n", addr_sp);
```

Early board and peripherals initialization.

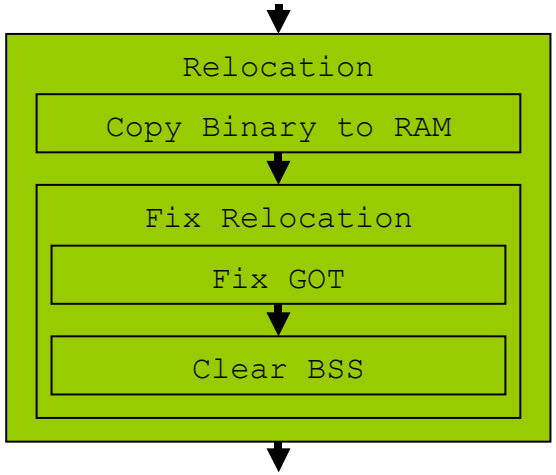
- Calculate the reserved memory areas and the stack address for general relocation. (cont.)
  - Finally we setup bank size for DRAM.
  - Write relocation addr, stack starting address, and the relocation offset into global data structure.
  - Finally call general relocation function “relocate_code()”.

arch/nds32/lib/board.c

```
   gd->bd->bi_baudrate = gd->baudrate;
        /* Ram isn't board specific, so move it to board
     code ... */
        dram_init_banksize();
        display_dram_config();  /* and display it */

        gd->relocaddr = addr;
        gd->start_addr_sp = addr_sp;

        gd->reloc_off = addr - _TEXT_BASE;

        debug("relocation Offset is: %08lx\n", gd-
     >reloc_off);

        memcpy(id, (void *)gd, GENERATED_GBL_DATA_SIZE);

        relocate_code(addr_sp, id, addr);

        /* NOTREACHED - relocate_code() does not return
     */
}
```

General Relocation.

- General relocation is the last binary copy of u-boot during bootstrap process.
  - Whether the u-boot is copied from ROM, FLASH, RAM or somewhere else, it general relocation will copy the u-boot into DRAM and to linking address adjustment.
  - Then the u-boot is ready to run in the RAM and responsible for loading OS image into RAM.



General Relocation.

- relocate_code:
  - First we save the parameters passed in for later calculation.
  - relocate_code(addr_sp, id, addr);
    - addr_sp: stack address
    - addr: relocation destination address.

arch/nds32/cpu/n1213/start.S:

```
/*
 * void relocate_code (addr_sp, gd, addr_moni)
 *
 * This "function" does not return, instead it
    continues in RAM
 * after relocating the monitor code.
 *
 */
.globl  relocate_code
relocate_code:
        /* save addr_sp */
        move    $r4, $r0
        /* save addr of gd */
        move    $r5, $r1
        /* save addr of destination */
        move    $r6, $r2
```

General Relocation.

- relocate_code:
  - Setup Stack.
  - Do binary copy from _start to __bss_start.
    - Calculate the offset between _start and __bss_start.

```
arch/nds32/cpu/n1213/start.S:

/* Set up the stack */
stack_setup:
        move    $sp, $r4

        la      $r0, _start

        beq     $r0, $r6, clear_bss     /* skip
    relocation */

        move    $r1, $r6                /* r1 <- scratch
    for copy_loop */
        la      $r3, __bss_start
        sub     $r3, $r3, $r0           /* r3 <-
    __bss_start_ofs */
        add     $r2, $r0, $r3           /* r2 <- source
    end address */
```

General Relocation.

- relocate_code:
  - Do binary copy from _start to __bss_start. (cont.)
    - Copy loop.
  - fix_relocations
    - Do address adjustment and linking address fix up.
      - Specify the starting address for fix.
      - Calculate the offset of relocation address where to fix.
      - $r9 will used in the following relocation fix.

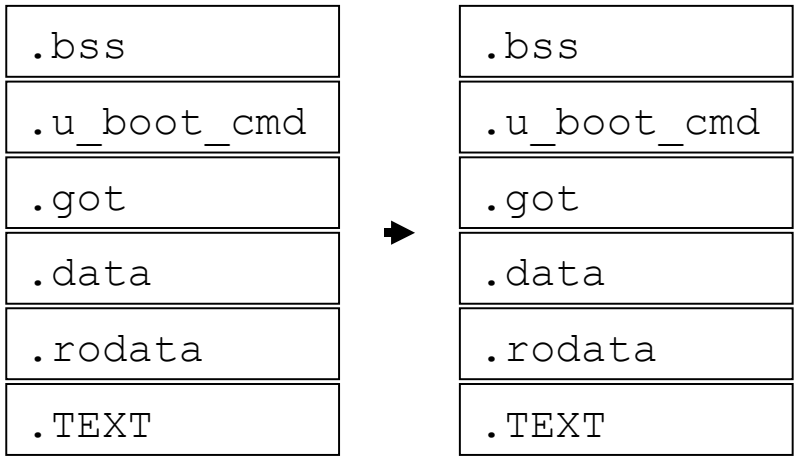
```
arch/nds32/cpu/n1213/start.S:

copy_loop:
        lwi.p   $r7, [$r0], #4
        swi.p   $r7, [$r1], #4
        blt     $r0, $r2, copy_loop

/*
 * fix relocations related issues
 */
fix_relocations:
        /* r0 <- Text base */
        l.w     $r0, _TEXT_BASE
        /* r9 <- relocation offset */
        sub     $r9, $r6, $r0
```

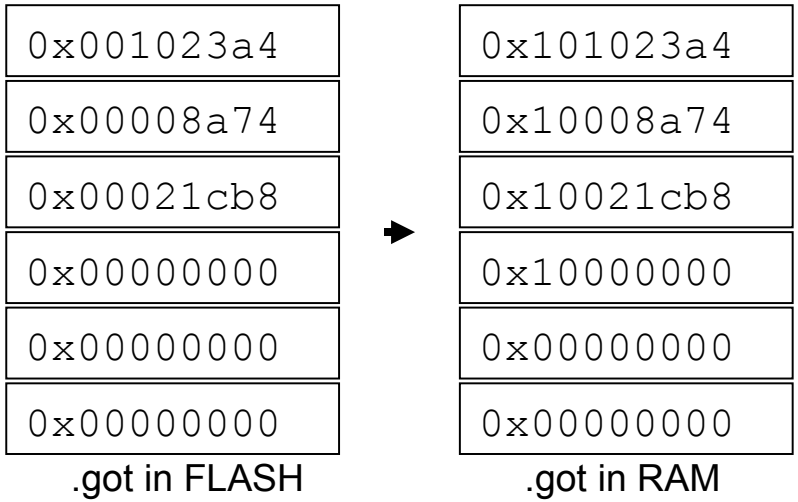
General Relocation.

- relocate_code:
  - fix_got:
    - GOT is **global offset table**, which is used for address lookup with $GP when indexing data and functions.
      - When the binary of u-boot is packed, the value of GOT was referenced by FLASH or ROM because u-boot ran in FLASH at the beginning.
      - Skip the first 2 entries since they must be zero.
  - Specify the starting address of got in FLASH and in RAM.
  - Specify the ending address of got in FLASH and in RAM.
  - Fix all value in GOT by relocation offset value in $r9.

```
arch/nds32/cpu/n1213/start.S:

fix_got:
/*
 * Now we want to update GOT.
 *
 * GOT[0] is reserved. GOT[1] is also reserved for the
    dynamic object
 * generated by GNU ld. Skip these reserved entries from
    relocation.
 */
        /* r2 <- rel __got_start in FLASH */
        la      $r2, __got_start
        /* r2 <- rel __got_start in RAM */
        add     $r2, $r2, $r9
        /* r3 <- rel __got_end in FLASH */
        la      $r3, __got_end
        /* r3 <- rel __got_end in RAM */
        add     $r3, $r3, $r9
        /* skipping first two entries */
        addi    $r2, $r2, #8
fix_got_loop:
        lwi     $r0, [$r2]              /* r0 <-
    location in FLASH to fix up */
        add     $r0, $r0, $r9           /* r0 <-
    location fix up to RAM */
        swi.p   $r0, [$r2], #4          /* r0 <- store
    fix into .got in RAM */
        blt     $r2, $r3, fix_got_loop
```

General Relocation.






```
nds32le-linux-objdump -D u-boot: (find __got_start)

000214d0 <__got_start>:
        ... <-- means 0x00
   214dc:       b8 1c           lwi37 $r0,[$fp+#0x70]
   214de:       02 00 74 8a     lhi $r0,[$r0+#-5868]
   214e2:       00 00 a4 23     lbi $r0,[$r1+#9251]
   214e6:       01 00 48 9c     lbi $r16,[$r0+#-14180]
   214ea:       01 00 74 16     lbi $r16,[$r0+#-3050]
   214ee:       02 00 40 17     lhi $r0,[$r0+#-32722]

hexdump -C uboot.bin:
000214d0  0000 0000 0000 0000  0000 0000 b81c 0200
000214e0  748a 0000 a423 0100  489c 0100 7416 0200
000214f0  4017 0000 880d 0000  b49c 0100 9c70 0000
00021500  b022 0100 ac1c 0200  00d4 0000 8897 0000

nds32le-linux-objdump -D u-boot:
00000f18 <board_init_r>:
     f18:       3a 6f a1 bc     smw.adm $r6,[$sp],
     $r8,#0x6
     f1c:       45 d2 07 58     movi $gp,#132952
     f20:       42 ff 80 20     mfusr $r15,$pc
     f24:       41 d7 f4 00     add $gp,$r15,$gp
     ...
     f34:       04 2e ff d6     lwi $r2,[$gp+#-168]
     f38:       04 3e ff f5     lwi $r3,[$gp+#-44]
     f3c:       58 00 00 01     ori $r0,$r0,#0x1
```

General Relocation.

- relocate_code:
  - clear_bss:
    - Clear BSS section to zero since it might have random value in RAM.

```
arch/nds32/cpu/n1213/start.S:

clear_bss:
        /* r0 <- rel __bss_start in FLASH */
        la      $r0, __bss_start

        /* r0 <- rel __bss_start in FLASH */
        add     $r0, $r0, $r9

        /* r1 <- rel __bss_end in RAM */
        la      $r1, __bss_end__

        /* r0 <- rel __bss_end in RAM */
        add     $r1, $r1, $r9

        li      $r2, 0x00000000    /* clear */

clbss_l:
        /* clear loop... */
        sw      $r2, [$r0]
        addi    $r0, $r0, #4
        bne     $r0, $r1, clbss_l
```

Relocation.

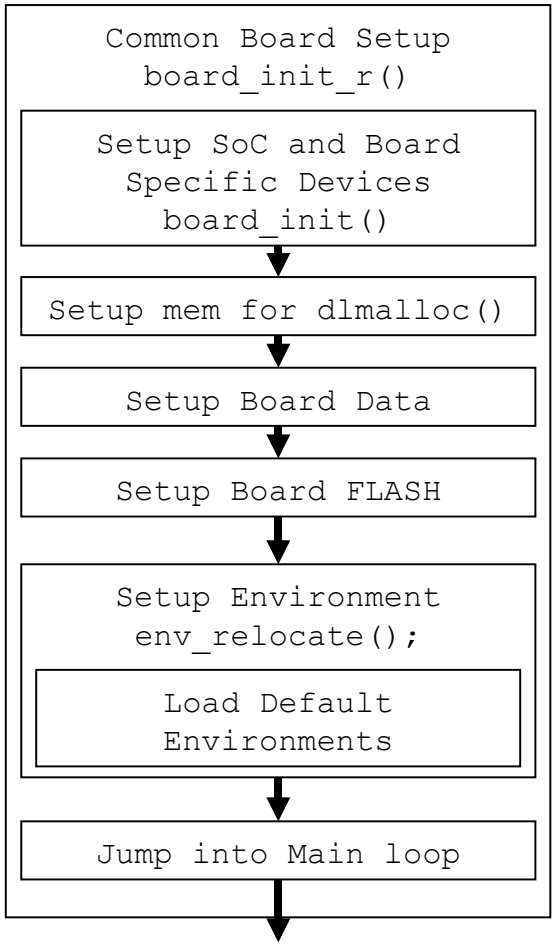
- relocate_code:
  - call_board_init_r:
    - Prepare to call common board setup function board_init_r().
      - Load the function address which the value was referenced in ROM.
      - Calculate the real address of board_init_r() in RAM.
      - Prepare parameters to call board_init_r().
    - Setup $r15 and $lp to jump into C environment.

arch/nds32/cpu/n1213/start.S:

```
/*
 * We are done. Do not return, instead branch to second
     part of board
 * initialization, now running from RAM.
 */
call_board_init_r:
        la      $r0, board_init_r

        /* offset of board_init_r() */
        move    $lp, $r0

        /* real address of board_init_r() */
        add     $lp, $lp, $r9

        /* setup parameters for board_init_r */
        move    $r0, $r5                /* gd_t */
        move    $r1, $r6                /* dest_addr */

#ifdef __PIC__
#ifdef __NDS32_N1213_43U1H__            /* NDS32 V0 ISA
     */
        move    $r15, $lp               /* store
     function address into $r15 */
#endif
#endif


        /* jump to it ... */
        jr      $lp                     /* jump to
     board_init_r() */
```

Common board and other peripherals initialization.

- Common board setup includes
  - SoC and customized board specific setup by calling board_init().
  - Memory setup for malloc() management.
  - Setup enviroment parameters.
  - Other peripherals devices.



Common board and other peripherals initialization.

- board_init_r()
  - Set GD_FLG_RELOC to indicate the relocation has been done.
  - Calculate the u-boot binary size in FLASH or ROM.
  - Call board_init() which will setup devices and for a specific SoC and board.

arch/nds32/lib/board.c:

```
/*
 * This is the next part if the initialization
    sequence: we are now
 * running from RAM and have a "normal" C
    environment, i. e. global
 * data can be written, BSS has been cleared, the
    stack size in not
 * that critical any more, etc.
 */
void board_init_r(gd_t *id, ulong dest_addr)
{
        char *s;
        bd_t *bd;
        ulong malloc_start;

        extern void malloc_bin_reloc(void);

        gd = id;
        bd = gd->bd;

        /* tell others: relocation done */
        gd->flags |= GD_FLG_RELOC;

        monitor_flash_len = &_end - &_start;
        debug("monitor flash len: %08lX\n",
    monitor_flash_len);

        board_init();   /* Setup chipselects */
```

Common board and other peripherals initialization.

- board_init_r()
  - fixup_cmdtable
    - Fix the u-boot command table relocation.
  - Calculate and setup the preserved memory space for dlmalloc (malloc management).
    - malloc_bin_reloc() will fix the relocation issue of the dlmalloc management table (bin) which is a bi-direction linking list maintain free spaces and addresses.
    - This part is important because cmd_env will use hash and malloc to management environment parameters.

```
arch/nds32/lib/board.c:
#if defined(CONFIG_NEEDS_MANUAL_RELOC)
        /*
         * We have to relocate the command table
     manually
         */
        fixup_cmdtable(&__u_boot_cmd_start,
                (ulong)(&__u_boot_cmd_end -
     &__u_boot_cmd_start));
#endif /* defined(CONFIG_NEEDS_MANUAL_RELOC) */

#ifdef CONFIG_SERIAL_MULTI
        serial_initialize();
#endif

        debug("Now running in RAM - U-Boot at: %08lx\n",
     dest_addr);

        /* The Malloc area is immediately below the
     monitor copy in DRAM */
        malloc_start = dest_addr - TOTAL_MALLOC_LEN;
        mem_malloc_init(malloc_start, TOTAL_MALLOC_LEN);
        malloc_bin_reloc();
```

Common board and other peripherals initialization.

- board_init_r()
  - Setup FLASH.
    - Do flash_init()
    - Calculate the flash offset and size.

```
arch/nds32/lib/board.c:
#ifndef CONFIG_SYS_NO_FLASH
        /* configure available FLASH banks */
        gd->bd->bi_flashstart =
    CONFIG_SYS_FLASH_BASE;
        gd->bd->bi_flashsize = flash_init();
        gd->bd->bi_flashoffset =
    CONFIG_SYS_FLASH_BASE + gd->bd->bi_flashsize;

        if (gd->bd->bi_flashsize)
                        display_flash_config(gd-
    >bd->bi_flashsize);
#endif /* CONFIG_SYS_NO_FLASH */
```

Common board and other peripherals initialization.

- board_init_r()
  - Setup the rest of devices.
    - Setup NAND flash.
    - Setup IDE devices.
    - Setup MMC/SD devices.
    - Setup PCI bridge and related devices.

```
#if defined(CONFIG_CMD_NAND)
        puts("NAND:  ");
        nand_init();            /*
   go init the NAND */
#endif

#if defined(CONFIG_CMD_IDE)
        puts("IDE:   ");
        ide_init();
#endif

#ifdef CONFIG_GENERIC_MMC
        puts("MMC:   ");
        mmc_initialize(gd->bd);
#endif

#if defined(CONFIG_CMD_PCI) ||
   defined(CONFIG_PCI)
        puts("PCI:   ");
        nds32_pci_init();
#endif
```

Common board and other peripherals initialization.

- board_init_r()
  - Setup the rest environments and functions.
    - Setup environments and import values from the environment structure stored in FLASH or ROM.
    - Get IP address from environments.
    - Setup API environment for standalone applications.
    - Setup remain function related to architecture extended functions or SoC.
    - Setup interrupt function.

```
arch/nds32/lib/board.c:
        /* initialize environment */
        env_relocate();

        /* IP Address */
        gd->bd->bi_ip_addr = getenv_IPaddr("ipaddr");

        /* get the devices list going. */
        stdio_init();

        jumptable_init();

#if defined(CONFIG_API)
        /* Initialize API */
        api_init();
#endif

        /* fully init console as a device */
        console_init_r();

#if defined(CONFIG_ARCH_MISC_INIT)
        /* miscellaneous arch dependent initialisations */
        arch_misc_init();
#endif
#if defined(CONFIG_MISC_INIT_R)
        /* miscellaneous platform dependent initialisations */
        misc_init_r();
#endif

#if defined(CONFIG_USE_IRQ)
        /* set up exceptions */
        interrupt_init();
        /* enable exceptions */
        enable_interrupts();
#endif
```

Common board and other peripherals initialization.

- board_init_r()
  - Setup the rest environments and functions.
    - Get load_address of OS image from environments.
    - Setup ethernet.
    - Finally we jump to main_loop() and waiting for commands from console.

```
arch/nds32/lib/board.c:
        /* Initialize from environment */
        load_addr = getenv_ulong("loadaddr", 16, load_addr);

#if defined(CONFIG_CMD_NET)
        s = getenv("bootfile");
        if (s != NULL)
                copy_filename(BootFile, s, sizeof(BootFile));
#endif

#ifdef BOARD_LATE_INIT
        board_late_init();
#endif

#if defined(CONFIG_CMD_NET)
        puts("Net:   ");

        eth_initialize(gd->bd);
#if defined(CONFIG_RESET_PHY_R)
        debug("Reset Ethernet PHY\n");
        reset_phy();
#endif
#endif

        /* main_loop() can return to retry autoboot, if so just
      run it again. */
        for (;;)
                main_loop();

        /* NOTREACHED - no way out of command loop except
      booting */
}
```

Summary

- Most of the configurable parameters related to architecture of a SoC are also listed in configuration file.
  - This provide you the flexibility to customize your system according to the application.
- Use lowlevel_init and remap correctly according to your design of the system.
- The bootstrap procedure is only the back ground knowledge for you before you start the porting task of a SoC.
  - For more information, please read the other slides.
    - U-boot porting guide for SoC.
- The bootstrap procedure will be improved continuously, so please follow up the most recent develop discussion on mailing list.

Appendix:
An Example of Vector Implements

```
 .align  5
tlb_fill:
        SAVE_ALL
        ! To get the kernel stack
        move    $r0, $sp
        ! Determine interruption type
        li      $r1, 1
        bal     do_interruption

        .align  5
tlb_not_present:
        SAVE_ALL
        ! To get the kernel stack
        move    $r0, $sp
        ! Determine interruption type
        li      $r1, 2
        bal     do_interruption

        .align  5
tlb_misc:
        SAVE_ALL
        ! To get the kernel stack
        move    $r0, $sp
        ! Determine interruption type
        li      $r1, 3
        bal     do_interruption
```

```
 .align  5
tlb_vlpt_miss:
        SAVE_ALL
        ! To get the kernel stack
        move    $r0, $sp
        ! Determine interruption type
        li      $r1, 4
        bal     do_interruption

        .align  5
machine_error:
        SAVE_ALL
        ! To get the kernel stack
        move    $r0, $sp
        ! Determine interruption type
        li      $r1, 5
        bal     do_interruption

        .align  5
debug:
        SAVE_ALL
        ! To get the kernel stack
        move    $r0, $sp
        ! Determine interruption type
        li      $r1, 6
        bal     do_interruption
```

Appendix: reset_cpu

- Do not use it.
- reset_cpu:
  - This macro was connect to do_reset but currently is obsoleted now.
    - The macro will reset CPU and jump to somewhere to execute a new binary.
    - Keep this only for reference, should be removed later.
  - The function do_reset will be called when you execute “reset” command in console.
    - Any kind of “reset” in a boot loader should be hardware reset to avoid hardware unstable includes dirty contents in the memory.

```
arch/nds32/cpu/n1213/start.S:
/*
 * void reset_cpu(ulong addr);
 * $r0: input address to jump to
 */
.globl reset_cpu
reset_cpu:
/* No need to disable MMU because we never enable it */

        bal     invalidate_icac
        bal     invalidate_dcac
        mfsr    $p0, $MMU_CFG
        andi    $p0, $p0, 0x3           ! MMPS
        li      $p1, 0x2                ! TLB MMU
        bne     $p0, $p1, 1f
        tlbop   flushall                ! Flush TLB
1:
        ! Get the $CACHE_CTL reg
        mfsr    $p0, MR_CAC_CTL
        li      $p1, DIS_DCAC
        ! Clear the DC_EN bit
        and     $p0, $p0, $p1
        ! Write back the $CACHE_CTL reg
        mtsr    $p0, MR_CAC_CTL
        ! Jump to the input address
        br      $r0
```