SISTEMI EMBEDDED
AA 2011/2012

Nios II processor

Reducing code size

Controlling code size (1)

- Very important to reduce memory costs
- The HAL environment includes only the features used by the application
  - If the Nios II hardware system contains exactly the peripherals used by the application, the HAL contains only the drivers necessary to control the hardware

Controlling code size (2)

- Available options to reduce code footprint (size)
  - Compiler optmisation
    - Some optimisation flags which control the trade-off between increasing speed and reducing memory use
  - Reduced device driver
    - Lighter device driver version (slower and less functions)

Peripheral	Small Footprint Behavior
UART	Polled operation, rather than IRQ-driven
JTAG UART	Polled operation, rather than IRQ-driven
Common flash interface controller	Driver excluded in small footprint mode
LCD module controller	Driver excluded in small footprint mode
EPCS serial configuration device	Driver excluded in small footprint mode

Controlling code size (3)

- Available options to reduce code footprint (size)
  - Reduce the File Descriptor Pool
    - The file descriptors that access character mode devices and files are allocated from a file descriptor pool. It can be changed through a BSP setting. The default is 32
  - Use /dev/null
    - At boot time, standard input, standard output, and standard error are all directed towards the null device, that is, /dev/null. After all drivers are installed, these streams are redirected to the channels configured in the HAL
    - The footprint of the code that performs this redirection is small, but you can eliminate it entirely by selecting null for stdin, stdout, and stderr when stdio is not used
    - You can control the assignment of stdin, stdout, and stderr channels by manipulating BSP settings

Controlling code size (4)

- Available options to reduce code footprint (size)
  - Use the Small newlib C Library. Some limitations:
    - No floating-point support for printf() family of routines
    - No support for scanf() family of routines
    - No support for seeking
    - No support for opening/closing FILE *. Only pre-opened stdout, stderr, and stdin are available
    - No buffering of stdio.h output routines
    - No stdio.h input routines
    - …
  - Use UNIX-Style File I/O fully omitting the C library
    - Standard I/O C functions can be emulated by application code

Controlling code size (5)

- Available options to reduce code footprint (size)
  - Use the Minimal Character-Mode API
    - If you can limit your use of character-mode I/O to very simple features, you can reduce code footprint by using the minimal character-mode API
    - This API includes the following functions:
      - alt_printf()
      - alt_putchar()
      - alt_putstr()
      - alt_getchar()
    - These functions are appropriate if the program only needs to accept command strings and send simple text messages.

Memory usage

Corresponds to virtual memories where the linker place code, data, stack, heap,...

Main | Software Packages | Drivers | Linker Script | Enable File Generation | Target BSP Directory

Linker Section Mappings

Linker Section Name	Linker Region Name	Memory Device Name
.bss	SDRAM	SDRAM
.entry	reset	SDRAM
.exceptions	SDRAM	SDRAM
.heap	SDRAM	SDRAM
.rodata	SDRAM	SDRAM
.rwdata	SDRAM	SDRAM
.stack	SDRAM	SDRAM
.text	SDRAM	SDRAM

Mapping

Linker Memory Regions

Linker Region Name	Address Range	Memory Device Name	Size (bytes)	Offset (bytes)
Onchip_memory	0x09000000 - 0x09001FFF	Onchip_memory	8192	0
SRAM	0x08000000 - 0x0807FFFF	SRAM	524288	0
SDRAM	0x00000020 - 0x007FFFFF	SDRAM	8388576	32
reset	0x00000000 - 0x0000001F	SDRAM	32	0

Corresponds to physical memories created w/ SoPC Builder.

Automatic code placement

- The reset handler code is always placed at the base of the *.reset* partition. The general exception funnel code is always the first code in the section that contains the exception address. By default, the remaining code and data are divided into the following output sections:
  - .text All remaining code
  - .rodata The read-only data
  - .rwdata Read-write data
  - .bss Zero-initialized data

Manually-controlled placement

- In your program source code, you can specify a target memory section for each piece of code. In C or C++, you can use the section attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".ext_ram.rwdata"))) = 0;
void bar (void) __attribute__ ((section (".sdram.txt")));
void bar (void)
{
foo++;
}
```

Stack and heap placement

- By default, the heap and stack are placed in the same memory partition as the .rwdata section
- The stack grows downwards (toward lower addresses) from the end of the section
- The heap grows upwards from the last used memory in the .rwdata section
- You can control the placement of the heap and stack by manipulating BSP settings
- By default, the HAL performs no stack or heap checking. This makes function calls and memory allocation faster, but it means that malloc() (in C) and new (in C++) are unable to detect heap exhaustion
- You can enable run-time stack checking by manipulating BSP settings. With stack checking on, malloc() and new() can detect heap exhaustion
- Stack checking has performance costs. If you choose to leave stack checking turned off, you must code your program so as to ensure that it operates within the limits of available heap and stack memory