Runtime Environment

Prof. Ken Short

Agenda

- Runtime environment
- Startup and exit code
- Where execution terminates
- Location of tables
- Interrupt service routines

The Runtime Environment (RTE)

- The *runtime environment* is the environment in which your application executes
- The runtime environment depends on the target hardware, the software environment, and the application code
- Either the IAR DLIB or CLIB runtime environment can be used as is together with the IAR C-SPY Debugger
- However, in some cases, to be able to run an application on hardware, you may have to adapt the runtime environment

Creating a Runtime Environment

- To create the required runtime environment a runtime library must be chosen and library options set

Runtime Library

- The runtime environment includes the *runtime library*, which contains the functions defined by the ISO/ANSI C and C++ standards, and include files that define the library interface
- There are two different runtime libraries provided:
  - The IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, et cetera.
  - The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. It does not fully support floating-point numbers in IEEE 754 format nor does it support Embedded C++. (**This is the default library and the one used in this course**).

RTE Support for the Target System

- The runtime environment also consists of a part with specific support for the target system, which includes:
  - Support for hardware features:
    - Direct access to low-level processor operations by means of *intrinsic* functions, such as functions for register handling
    - Peripheral unit registers and interrupt definitions in include files
    - Special compiler support for accessing strings in flash memory, see *AVR–specific library functions*
  - Runtime environment support, that is, startup and exit code and low-level interfaces to some library functions.
- Some parts, like the startup and exit code and the size of the heaps must be tailored for the specific hardware and application requirements.

Assembly Language Projects

- If your project contains only assembly language source code there is no need for a runtime library

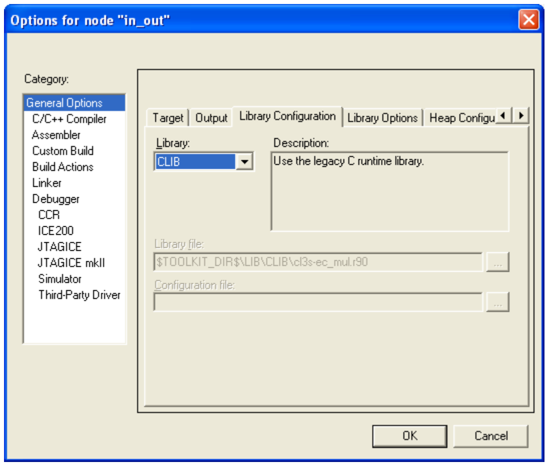
Runtime Library Selection

- To configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your resulting code.

- To get the most code efficient runtime environment, you can customize IAR's prebuilt runtime libraries by:
  - Setting library options, for example, for choosing scanf input and printf output formatters, and for specifying the size of the stack and the heap
  - Overriding certain library functions, for example cstartup.s90, with your own customized versions
  - Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibytes, by choosing a *library configuration*: normal or full.

Using the Default CLIB Library

- For our applications we will use the default CLIB library



2/15/2016 © Copyright Kenneth Short 2006 9

Providing the Missing OS Services

- When you execute a program on a general purpose computer certain services are provided transparently by the operating system
- The program to be executed gets loaded from disk into system's RAM. The stack pointer(s) for the program are initialized. The operating system then initializes all variables that require initialization to their initial values
- The flow of control is then transferred by the operating system to the program that is to be executed. For a C program this is to the first instruction in main()

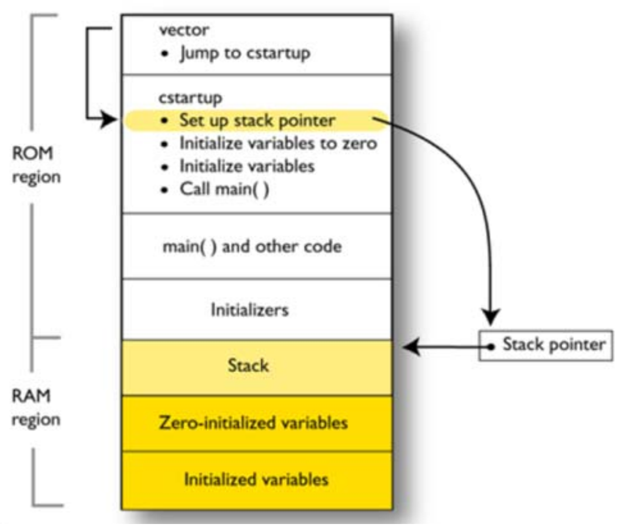
2/15/2016 © Copyright Kenneth Short 2006 10

Providing the Missing OS Services (cont.)

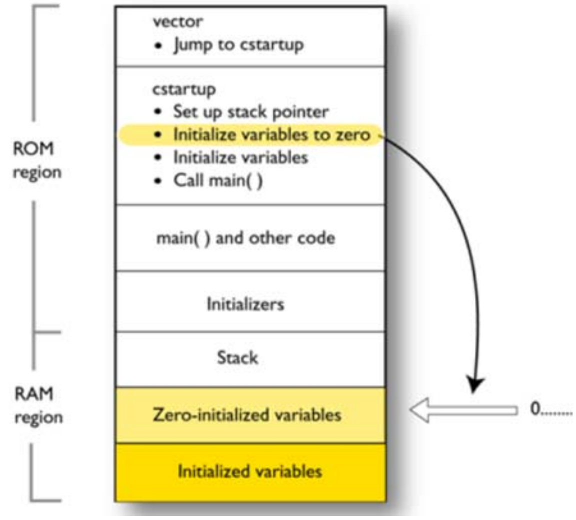
- In an embedded system implementation that does not have an operating system, these services must be implemented by startup code that is executed before control is passed to main()
- The startup code is automatically provided by IAR Embedded Workbench and linked into the final object code
- Accordingly, when the AVR microcontroller is reset, the reset vector does not transfer control to the first instruction in main() but rather to the first instruction in the startup code

2/15/2016 © Copyright Kenneth Short 2006 11

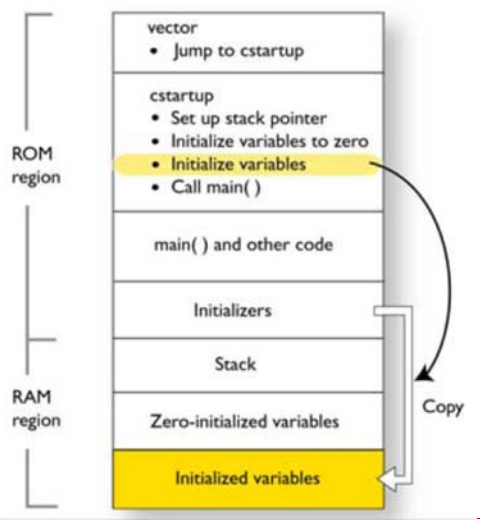
Set Up Stack Pointer



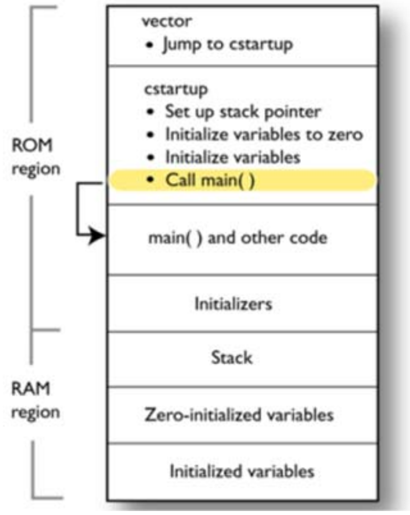
Zero Variables Initialized to 0



Initialize Non-zero Initialized Variables



Call main() Function



Startup and Exit Code

- The code for handling startup and termination is located in the source files cstartup.s90 and _exit.s90, and low_level_init.c located in the avr\src\lib directory.
- These files are automatically linked into your code when you build an application in the IAR IDE.
- For our work in this course we will not modify these files.

Startup and Exit Sequences

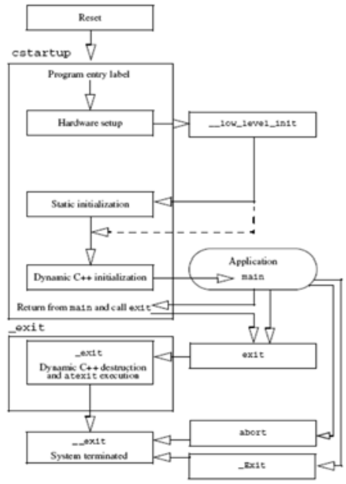


Figure 1: Startup and exit sequences

2/15/2016 © Copyright Kenneth Short 2006 17

System Startup Steps Using CLIB

- When the cpu is reset it jumps to the program entry label __program_start in the system startup code.
- The startup code:
  - Enables the external data and address buses if needed
  - Initializes the stack pointers to the end of RSTACK and CSTACK, respectively
  - Initializes static variables except for __no_init and __eeprom declared variables; this includes clearing zero-initialized memory and copying the ROM image of the RAM memory for the initialized variables
  - Calls custom function __low_level_init, giving the application a chance to perform early initializations prior to the start of main()
  - Calls main function, which starts the application

2/15/2016 © Copyright Kenneth Short 2006 18

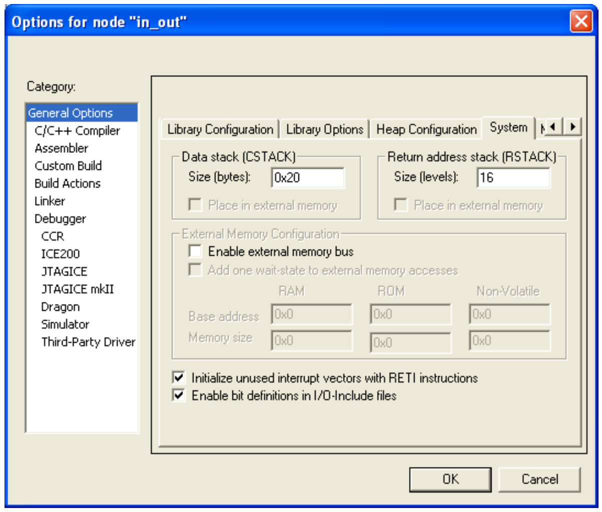
The Return Address Stack (RSTACK)

- The *return address stack* (RSTACK) and the *data stack* (CSTACK) are two separate stacks.
- The return address stack is used for storing the return address when a CALL, RCALL, ICALL, or EICALL instruction is executed. Each call will use two or three bytes of return address stack.
- An interrupt will also place a return address on this stack.
- The system startup code initializes SP to the end of the return address stack.

Data Stack (CSTACK)

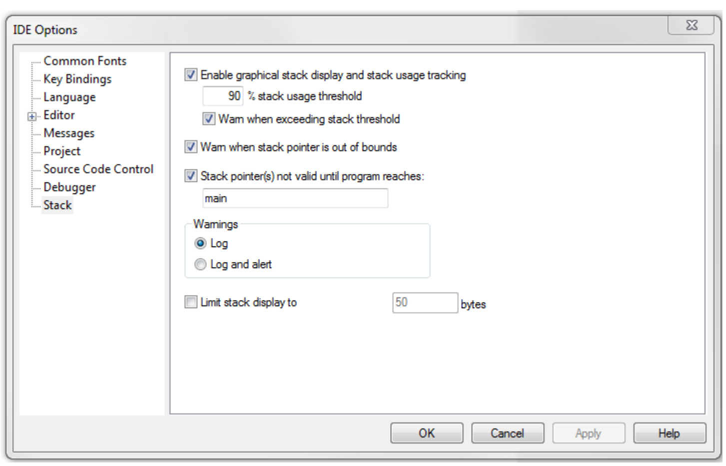
- The data stack is used by functions to store auto variables, function parameters and temporary storage that is used locally by functions.
- The data stack is a continuous block of memory pointed to by the AVR's pointer register Y.
- The data segment used for holding the stack is called CSTACK.
- The system startup code initializes pointer Y to the end of the data stack.

Setting the Stack Sizes



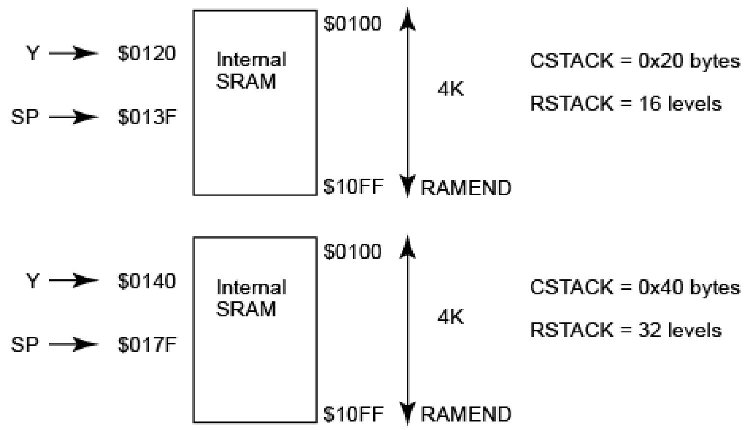
2/15/2016 © Copyright Kenneth Short 2006 21

Enabling Stack Usage Warning



2/15/2016 © Copyright Kenneth Short 2006 22

Example Stack Initializations



2/15/2016 © Copyright Kenneth Short 2006 23

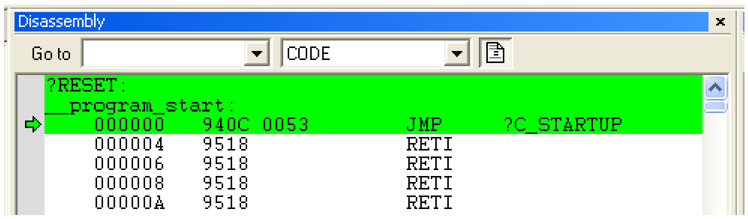
Startup Code for in_out Program

```
?C_FUNCALL:
_exit:
exit:
    0000A0    0000            NOP
?C_EXIT:
__exit:
    0000A2    9588            SLEEP
    0000A4    CFFE            RJMP    ?C_EXIT
?C_STARTUP:
__RESTART:
    0000A6    E30F            LDI     R16,0x3F
    0000A8    BF0D            OUT     SPL,R16
    0000AA    E001            LDI     R16,0x01
    0000AC    BF0E            OUT     SPH,R16
    0000AE    E2C0            LDI     R28,0x20
    0000B0    E0D1            LDI     R29,0x01
?call_low_level_init:
    0000B2    940E 0061       CALL    __low_level_init
?cstartup_call_main:
    0000B6    940E 0046       CALL    main
    0000BA    940E 0050       CALL    ?C_FUNCALL
    0000BE    940C 0050       JMP     ?C_FUNCALL
__low_level_init:
    0000C2    E001            LDI     R16,0x01
    0000C4    9508            RET
```

2/15/2016 © Copyright Kenneth Short 2006 24

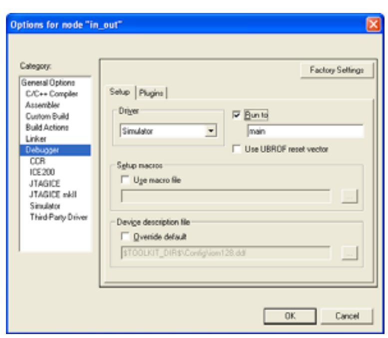
C-SPY at Reset

- At reset, a jmp to ?C_STARTUP is executed. This is label at the beginning of the startup code.
- This instruction created by the compiler



Where Execution Begins in C-SPY

- If **Run to main** is checked in Options > Debugger, C-SPY executes through the startup code and breaks at the first instruction in main(). As a result you don't normally see the execution of the startup code



System Termination

- An application can terminate normally in two different ways:
  - Return from the main function
  - Call the exit function.
- Because the ISO/ANSI C standard states that the two methods should be equivalent, the cstartup code calls the exit function if main returns. The parameter passed to the exit function is the return value of main. The default exit function is written in assembler.
- When the application is built in debug mode, C-SPY stops when it reaches the special code label ?C_EXIT.
- An application can also exit by calling the abort function. The default function just calls __exit in order to halt the system, without performing any type of cleanup.

Returning from main()

- There are two types of main functions, those that return and those that are infinite loops.
- For those that return, after they return code is executed that puts the MCU into sleep mode. If the MCU wakes from sleep mode it will just enter sleep mode again. This is useful for a completely interrupt driven program.
- If the program is completely interrupt driven the MCUCR register must be configured before the main function returns.
- If the program is not completely interrupt driven and needs to execute code in main when woken up. The main function must be an infinite loop and use the __sleep() intrinsic function.

Where Execution Terminates in C-SPY

- C-SPY stops executing when it reaches the special label ?C_EXIT

Next Class

Reading Assignment