Embedded Programming in C in a Nutshell ...

Today's Agenda

- A bit about everything ...

FPGA-based SoC Design

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob
University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

Embedded Programming in C in a Nutshell in a Nutshell

International Master of Science in Electrical Engineering

Prof. Dr. C. Jakob

University of Applied Sciences Darmstadt
h_da
Faculty of Electrical Engineering and Information Technology
fbeit

Motivation

- Up until today, C is the language of choice when programming embedded systems.
- The main advantage of C is its support for pointers
  - Allows memory/peripheral device access.
- The main disadvantage of C is its support for pointers
  - Main source of coding errors …
- So what's the conclusion?
  - Handle pointers with care!
  - Understand pointer arithmetic's!
  - Avoid deep indirection: pointer to pointer to pointers …
- We use C for hardware oriented programming

Motivation

- The use of pointers enables the access of hardware registers that have a specific address (location) in the memory map of the system.
- If necessary, it is possible to mix C and assembly code.
- We will several C features such as structs and functions to hide low-level code from the application programmer.

General Concepts

Embedded Programming in C in a Nutshell

General layout of the main() function

Board support package header file, typically by the µC vendor

```c
#include "bsp.h"

int main(void){

    system_init();

    while(1) {

        ... // Application code runs in an infinite loop …
    }
    return 0;
}
```

Different to the main function found in most textbooks: it doesn't take any arguments …

Configure and setup the various peripheral components such as GPIOs, ADCs, Timers, …

we should never get here …

[C] Source Code Snippet

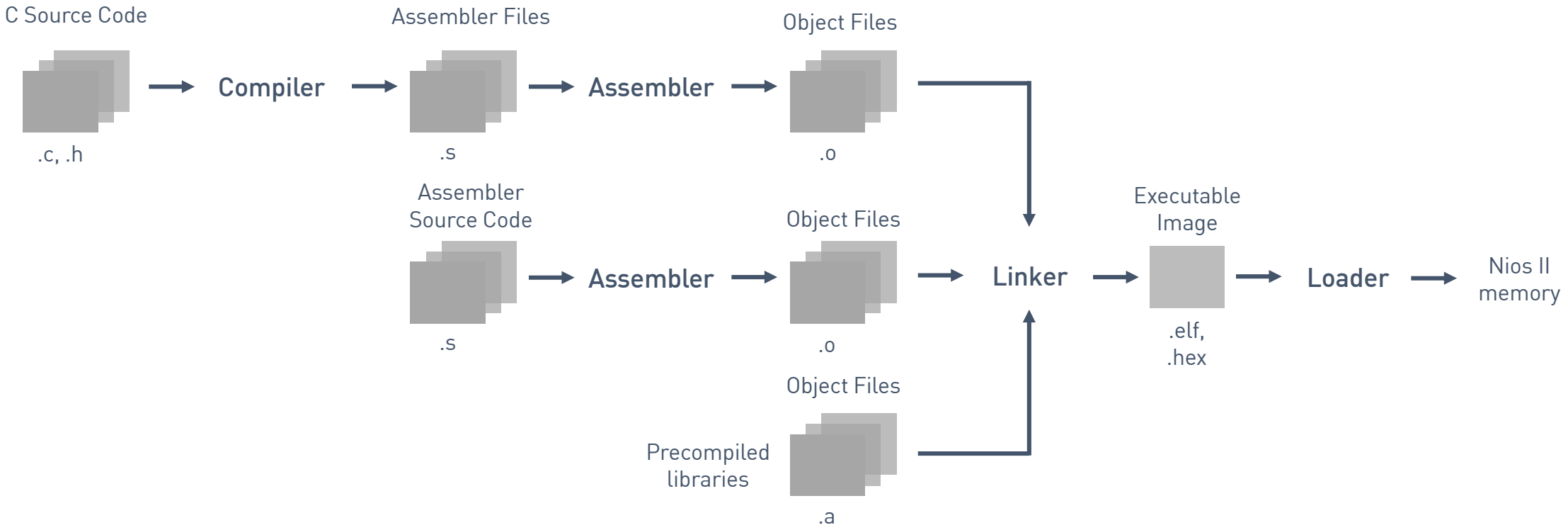
Notes

- A quick and useful hack …

Build Tools

Embedded Programming in C in a Nutshell

Build Tools


C Source Code
.c, .h
Compiler
Assembler Files
.s
Assembler
Object Files
.o
Assembler Source Code
.s
Assembler
Object Files
.o
Precompiled libraries
Object Files
.a
Linker
Executable Image
.elf, .hex
Loader
Nios II memory

Build Tools

```
1. nios2-elf-gcc -o main.elf main.s
```

Assemble and Link

```
1. nios2-elf-gcc -o main.elf main.c
```

Compile and Link

```
1. nios2-elf-g++ -o <your project>.elf <object files> -lm
```

Linker only

```
1. nios2-download -g main.elf
```

Download to target memory

```
1. nios2-elf-objdump -D -S -x <project>.elf > <project>.elf.objdump
```

Use this command to display information about the (.elf) object file

Build Tools

```
1. nios2-elf-readelf -symbols <project>.elf | grep <function name>
```

Display information about all instances of a specific function name in the .elf file

```
1. nios2-elf-ar q <archive_name>.a <object files>
```

This command generates an archive (.a) file containing a library of object (.o) files

```
1. nios2-elf-size my_project.elf
2. text data bss dec hex filename
3. 272904 8224 6183420 6464548 62a424 my_project.elf
```

This command displays the total size of your program and its basic code sections

```
1. nios2-elf-strings <project>.elf
```

This command displays all the strings in a .elf file

Build Tools

```
1. nios2-elf-objdump -D -S -x <project>.elf > <project>.elf.objdump
```

Use this command to display information about the object file

Loading Data and Code into Memory

Embedded Programming in C in a Nutshell

Loading Data and Code into Memory

- In general, any data is mapped to RAM while code is mapped to FLASH

Accessing Data in Memory

Embedded Programming in C in a Nutshell

Accessing Data in Memory

- Accessing data in memory requires a basic understanding about how memory is allocated in C programs.
- According to the GNU C documentation, the C language supports two kinds of memory allocation through the variables in C programs:
  - **Static allocation** is what happens when you declare a static or global variable. Each static or global variable defines one block of space, of a fixed size. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.
  - **Automatic allocation** happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound
- A third important kind of memory allocation, dynamic allocation, is not supported by C variables but is available via GNU C Library functions.
  - `malloc()` - Allocate dynamic memory

Accessing Data in Memory

```
1.  static alt_u32 var_1;                    // static global variable
2.
3.  alt_u32 var_2;                           // non-static global variable
4.
5.  int main(void){
6.  ...
7.
```

[C] Source Code Snippet

Notes

- Both type of variables are statically allocated at compile-time.
- So what is the difference between a **non-static global** and **static global** identifier in C?
  - non-static global variable: external linkage
  - static global variable: internal linkage
- **Internal linkage** refers to everything only in scope of a translation unit. **External linkage** refers to things that exist beyond a particular translation unit. In other words, accessible through the whole program, which is the combination of all translation units (or object files).

Accessing Data in Memory – Static Variables

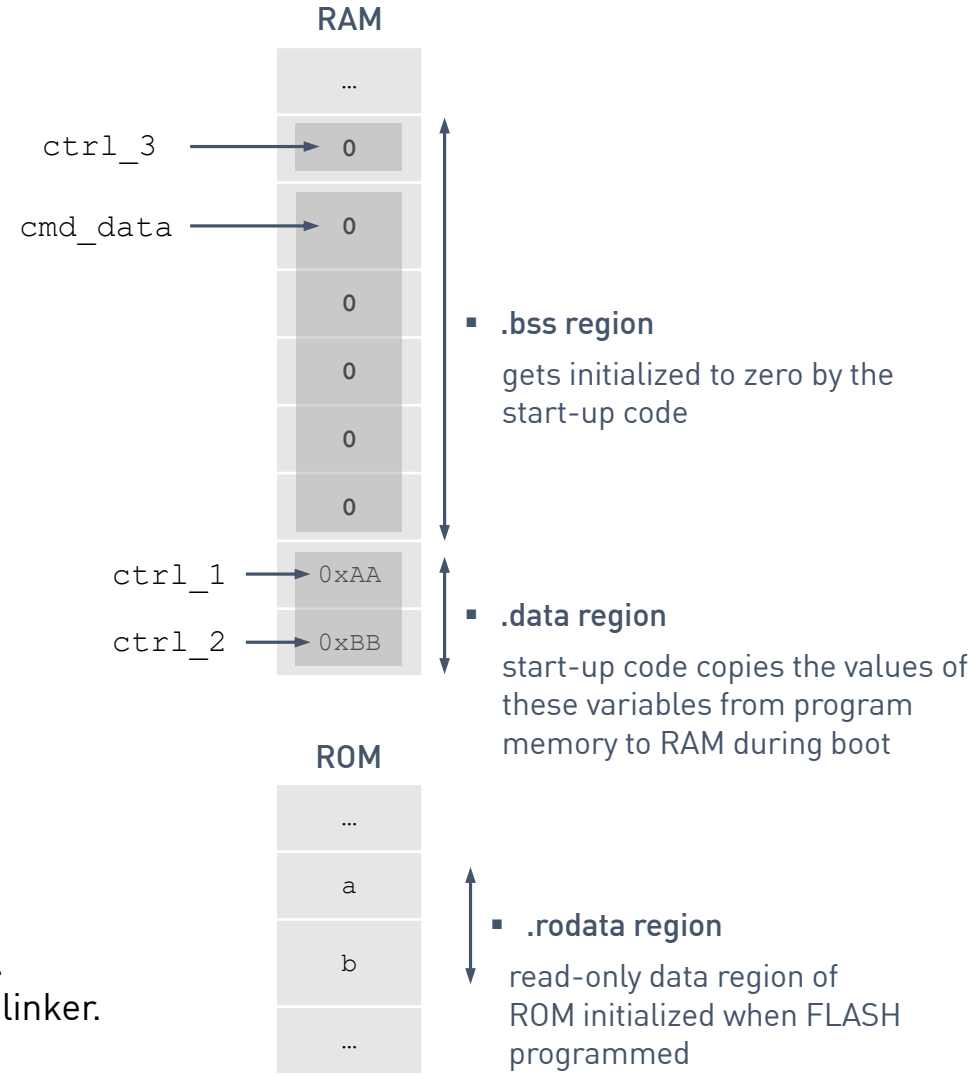
Embedded Programming in C in a Nutshell

Accessing Data in Memory – Static Variables

```
1.  static alt_u32 cmd_data[5];
2.
3.  // specify the initial values of static variables
4.  static alt_u32 ctrl_1 = 0xAA;
5.  static alt_u32 ctrl_2 = 0xBB;
6.
7.  const alt_u8 ctrl_args[2] = {'a','b'};
8.
9.  void set_ctrl_args(void){
10.     static alt_u32 ctrl_3;
11.
12. }
```

static lifetime: by default, a variable declared inside a function is an automatic variable

[C] Source Code Snippet



Notes

- Static variables declared **outside** of any function are global variables with limited visibility.
- Static variables declared **inside** of a function: Static lifetime and memory allocated by the linker.
- Memory sections for static variables are allocated by the linker.

Accessing Data in Memory – Static Variables

```
1.  nios2-elf-gcc -W1, -Map = main.map -O2 -g -o main.elf main.c
```

Notes

- The final memory map of the actual implementation can be analysed using the linker map.
- The linker map is generated as a by-product of the compilation process and shows the memory layout for function and static variable names (elements with external visibility).

name	address	size	object file
.text.main	0x0000100c	0x110	./Sources/main.o
	0x0000100c		main
.rodata.Digits			
	0x000025b4	0x4	./Sources/main.o
.data.varB	0x1fffe004	0x4	./Sources/main.o
.data.varC	0x1fffe008	0x4	./Sources/main.o
.bss.varA	0x1fffe11c	0x14	./Sources/main.o

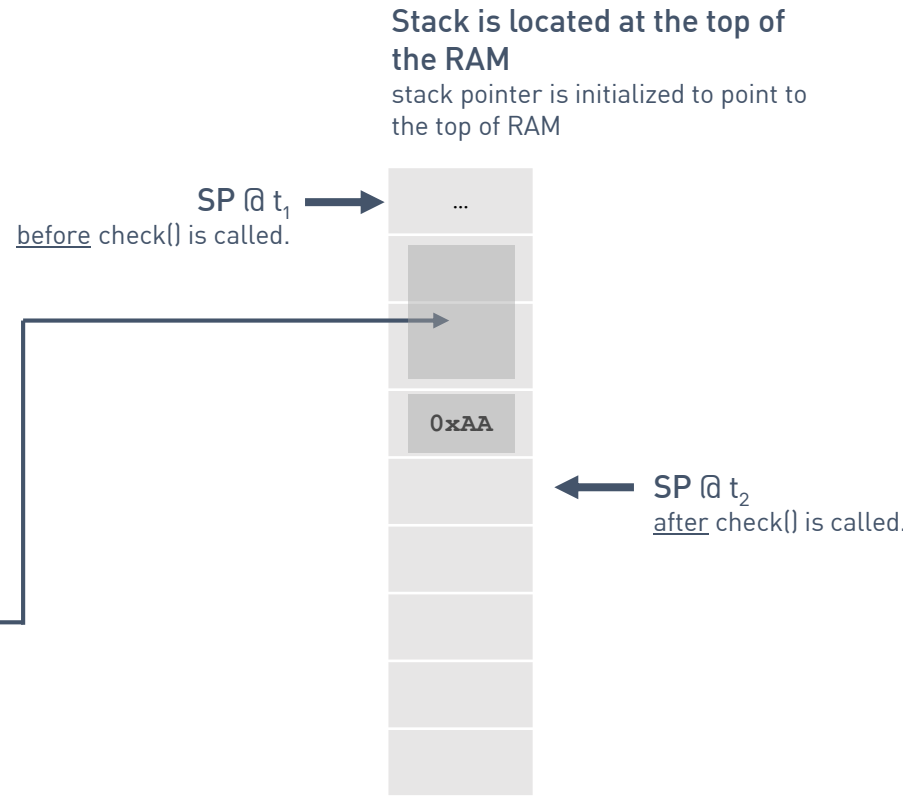
Accessing Data in Memory – Automatic Variables

Embedded Programming in C in a Nutshell

Accessing Data in Memory – Automatic Variables

```
1.  void check(void) {
2.      // local variable lives on the stack …
3.      alt_u32 tmp_code = 0xAA;
4.      ...
5.  }
6.
7.  int main(void){
8.
9.      ...
10.     check();
11.     ...
12. }
```

As functions get called out of main(), information needs to stored on the stack such as the **function return address** as well as all **registers** that needs to be saved …



[C] Source Code Snippet

Notes

- Variables declared in function are automatic variables, which are stored on the Stack.
- The stack memory is accessed using the stack pointer SP. The stack grows and shrinks automatically as variables are created and go out of scope.
- **Warning**: Be careful with **pointers to automatic variables**. If variables are no longer in scope, the stack location might be used by a different variable! Thus, the pointer would be invalid because it is pointing to a memory location which is no longer in use!

Accessing Data in Peripheral Registers

Embedded Programming in C in a Nutshell

Accessing Data in Peripheral Registers

known address .

```
1. volatile alt_u32 * pio_reg = (volatile alt_u32 *) 0x80009120;
2.
3. *pio_ptr = 0xFFFFFFFF;     // set all bits of the pio register to 1 ...
4.
```

dereference the pointer to access the register.

Notes

- Pointers provide access to any variable or peripheral register whose address is known.
- Please note that the pointer type must match the size of the object being addressed. In the upper example, we are accessing a 32-bit wide register. Therefore, the pointer type must be of type `alt_u32` (header file: `alt_types.h`)

Accessing Data in Peripheral Registers

```
1  alt_u32 * pio_reg = (alt_u32 *) 0x80009120;
2.
3. while((*pio_reg_ptr) & (1 << 5)) != 0) {
4.     // loop until bit 5 of pio_reg is set ...
5. }
```

Notes

- **Read-Only Register - What's the problem with the previous code if the pointer is not declared as volatile:** Compiler may optimize the code to only read the pio register once, because the program never writes to a read-only register! The variable would solely live in a processor register without being updated anymore … Therefore, the pointer should be declared as volatile to prevent optimization.

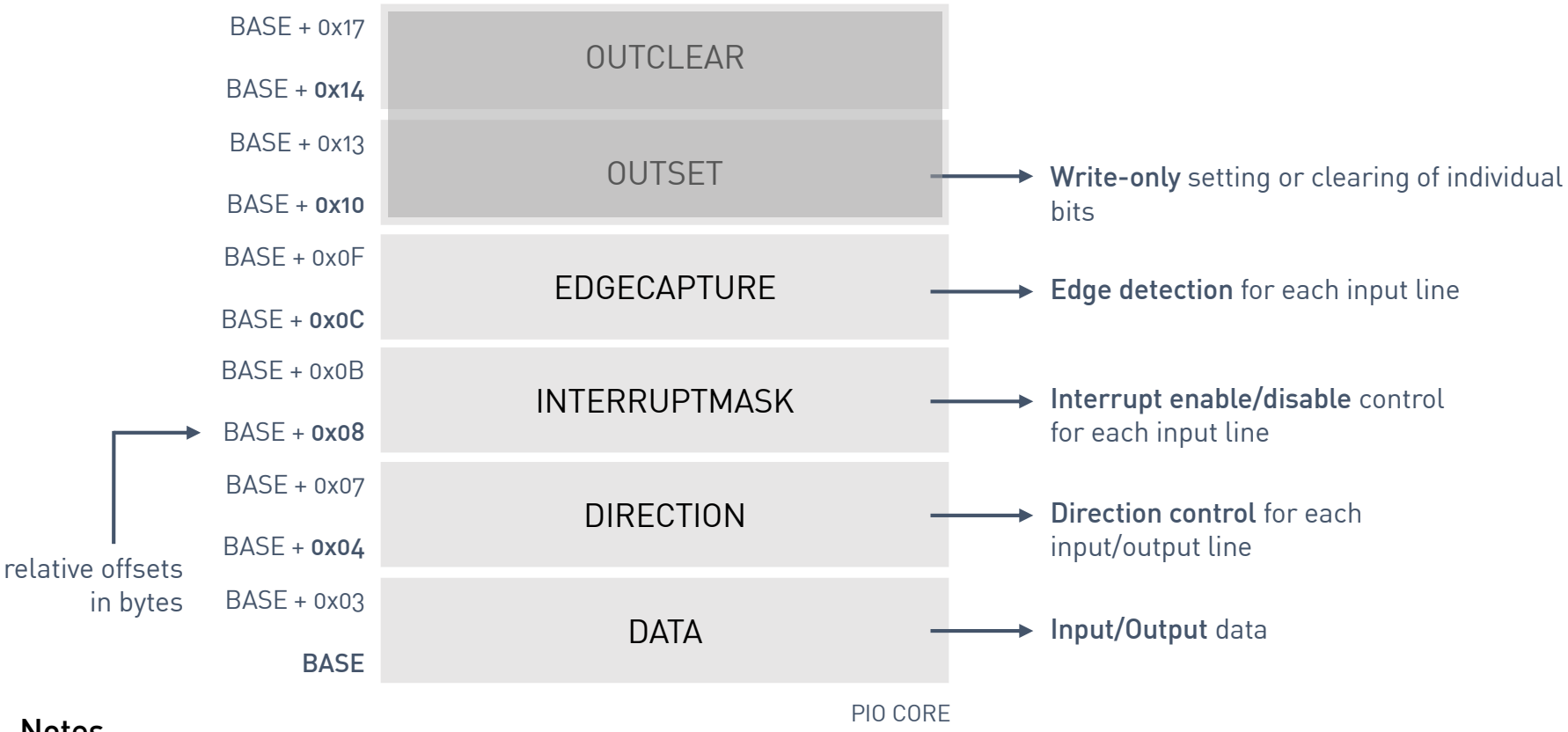
```
1  volatile alt_u32 * pio_reg = (volatile alt_u32 *) 0x80009120;
2.
3. while((*pio_reg_ptr) & (1 << 5)) != 0) {
4.     // loop until bit 5 of pio_reg is set ...
5. }
```

- The volatile keyword tells the compiler to assume the variable can be changed by another process or by a hardware device. Generally the compiler will not keep such variables in registers and will re-read them from memory whenever they are used. It will also store new values to memory whenever told to.

Accessing Data in Peripheral Registers

- Typically, registers are placed at sequential addresses in the memory map ...
- Struct members can match the memory map of peripheral registers.

Accessing Data in Peripheral Registers



Notes

- PIO core, Embedded Peripherals IP User Guide

Accessing Data in Peripheral Registers


```
1. #define __I    volatile const  // read only permission
2. #define __IO   volatile        // read/write permission
3. #define __O    volatile        // write only permission ;-) doesn't work in C...
4.
5. #define GPIO_BASE_ADDRESS 0x80090000
6.
7. typedef struct {
8.     __IO alt_u32 DATA_REG;
9.     __IO alt_u32 DIRECTION_REG;
10.    __IO alt_u32 INTERRUPTMASK_REG;
11.    __IO alt_u32 EDGECAPTURE_REG;
12.    __O  alt_u32 OUTSET_REG;
13.    __O  alt_u32 OUTCLEAR_REG;
14. }
15. PIO_TYPE;
16.
17. #define LEDS ((PIO_TYPE *) GPIO_BASE_ADDRESS))
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #4

Accessing Data in Peripheral Registers

1/1

```
1.  LEDS->DATA ^= 0xFF;
2.
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #4

Notes

- We use struct-pointer dereferencing to access the individual registers

Accessing Data in Peripheral Registers


```
1. #define __I    volatile const  // read only permission
2. #define __IO   volatile        // read/write permission
3. #define __O    volatile        // write only permission ;-) doesn't work in C...
4.
5. #define GPIO_BASE_ADDRESS 0x80090000
6.
7. typedef struct {
8.     __IO alt_u32 DATA_REG;
9.     __IO alt_u32 DIRECTION_REG;
10.    __IO alt_u32 INTERRUPTMASK_REG;
11.    __IO alt_u32 EDGECAPTURE_REG;
12.    __O  alt_u32 OUTSET_REG;
13.    __O  alt_u32 OUTCLEAR_REG;
14. }
15. PIO_TYPE;
16.
17. #define LEDS (*((PIO_TYPE *) GPIO_BASE_ADDRESS)))
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #5

Accessing Data in Peripheral Registers

1/1

```
1.  LEDS.DATA ^= 0xFF;
2.
```

[C] Source Code Excerpt: Nios II "Hello, World!“ – Version #4

Notes

- We can use the dot operator to access the output data register DATA.
- Both statements are equivalent and generate the same set of machine instructions.

Bit Manipulation in C

Embedded Programming in C in a Nutshell

Bit Manipulation in C

- The manipulation of individual bits is one of the most important and fundamental concepts to understand when programming microcontroller-based embedded systems.
- Bit manipulation is necessary to read the status of components, set parameters, and to change the state of output pins to name just a few.
- It is important to understand how to individually change the states of certain bits while leaving others unchanged.

Bit Manipulation in C

- The C language provides six bitwise operators for bit manipulation.
- These operators act on **integral operands** (char, short, int and long) represented as a string of binary digits.
- **Bitwise operators:**
  - bitwise complement: ~
  - bitwise AND: &
  - bitwise OR: |
  - bitwise XOR: ^
- **Shift operators:**
  - left shift <<
  - right shift >>

&=	
|=	
^=	
<<=	
>>=	

Shorthand operators

Bit Manipulation in C

- The manipulation of individual bits is one of the most important and fundamental concepts to understand when programming microcontroller-based embedded systems.
- Bit manipulation is necessary to read the status of components, set parameters, and to change the state of output pins to name just a few.
- It is important to understand how to individually change the states of certain bits while leaving others unchanged.

Something in between ...

Embedded Programming in C in a Nutshell

Binary Printf ...

```c
#include <stdio.h>

void printf_bin(unsigned char byte){
     char c [8+1];

     unsigned char i;
     for(i = 0; i < 8; i++){
          c[7-i] = (byte & (1 << i)) ? '1' : '0';
     }
     c[8] = '\0';      // Null-terminated string ...

     printf("%s \n",c);
}
```

[C] Source Code Snippet

Notes

- A quick and useful hack ...

Bit Manipulation in C contd.

Embedded Programming in C in a Nutshell

Bit Manipulation in C

- The concepts discussed on the next slides all refer to an 8-bit variable/register.
- This has been done for improving clarity.
- All concepts discussed in the following sections apply in the exact same manner to 32-bit variables/registers as well.
- With start with discussion bitmask in C.
- Motivation: **We are declaring a bitmask to identify the bits** (within a byte) **that we intend to manipulate.**
- Manipulation might refer to operations such as setting/clearing or inverting bits …

Bit Manipulation in C

```
1.  unsigned char bit_mask = 0b00100100;                    // set bits 2 & 5
```

Notes

- We are declaring a bitmask to label the bits (within a byte) that we intend to manipulate (set, delete, invert).
- The previous expression can be rewritten as shown below. This approach clearly highlights the bits to be manipulated.

```
1.  unsigned char bit_mask = (1 << 2) | ( 1 << 5);          // set bits 2 & 5
```

Notes

- The left-shift << operation reads as follows: The operation `x << n` shifts the value of `x` left by `n` bit positions.
- The previous expression in detail ...

```
1.   unsigned char bit_mask = (1 << 2) | ( 1 << 5);
2.
3.  (1 << 2)   ⟶   (0b00000001 << 2)   ⟶   0b 0000 0100
4.  (1 << 5)   ⟶   (0b00000001 << 5)   ⟶   0b 0010 0000
5.                                          ---------          // bitwise OR operation
6.                              bit_mask = 0b 0010 0100
```

- The bitmask is identical to the first one, however it cleary highlights the bits that we intend to manipulate ...

OR truth table

x	y	q
0	0	0
0	1	1
1	0	1
1	1	1

Bit Manipulation in C

- Maybe the advantage of this approach does not become clear for an 8-bit variable …
- Let's consider a 32-bit variable:

```
1.  unsigned int bit_mask = 0b00100100000001000010000000100000;
```

Notes

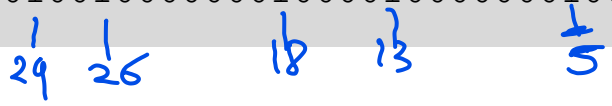
- Which bits are set in the previous example? ;-)

unsigned int bit_mask = (1<<5) | (1<<13) | (1<<18) | (1<<26) | (1<<29)

Bit Manipulation in C

- Maybe the advantage of this approach does not become clear for an 8-bit variable …
- Let's consider a 32-bit variable:

```
1.  unsigned int bit_mask = 0b00100100000001000010000000100000;
```



Notes

- Which bits are set in the previous example? ;-)

```
1.  unsigned int bit_mask = (1 << 5) | (1 << 13) | (1 << 18) | (1 << 26) | (1 << 29);
```

Bit Manipulation in C – Setting Individual Bits

Embedded Programming in C in a Nutshell

Setting Individual Bits

normally, the state of the variable/register to be manipulated is unknown ...

```
1. // variable to be manipulated
2. unsigned char my_variable = 0b10001000;
3. // bitmask - definition of the bit positions to be manipulated
4. unsigned char bit_mask = ((1 << 2) | (1 << 5));
5. // Setting bits using the bitwise OR operator
6. my_variable = my_variable | bitmask;
```

Notes

- The previous expression in detail ...

```
1. (1 << 2)   ⟶  (0b00000001 << 2)  ⟶  0b 0000 0100
2. (1 << 5)   ⟶  (0b00000001 << 5)  ⟶  0b 0010 0000
3.                                     ------------------   // bitwise OR (bitmask)
4.                                bit_mask = 0b 0010 0100
5.                             my_variable = 0b 1000 1000
6.                                     ------------------   // bitwise OR (variable)
7.                             my_variable = 0b 1010 1100
```

- Bits 2 and 5 are set, all other bits remain unchanged.

OR truth table

x	y	q
0	0	0
0	1	1
1	0	1
1	1	1

Setting Individual Bits

```
1. my_variable = my_variable | bitmask;
```

Notes

- Please note that the previous expression can also be expressed using the following shorthand assignment operator.

```
1. my_variable |= bitmask;
```

Notes

- The previous operation represents a **Read-Modify-Write** operation. This involves:

  1. **Reading** the state of the variable/register,
  2. **Modifying** the state of that variable/register by performing a bitwise logical operation on the value read from the variable/register, with a bitmask that specifies the bits to be manipulated, and finally,
  3. **Writing** back the result into the variable/register.

- In this way, only the bits of interest are modified, while all other bits do not change.
- The **RMW operations are not strictly atomic**, i.e. they will very likely not happen in a single instruction cycle and may even take several instruction cycles to execute. This requires special attenuation in various design scenarios. More on that later.

Bit Manipulation in C – Clearing Individual Bits

Embedded Programming in C in a Nutshell

Clearing Individual Bits

normally, the state of the variable/register to be manipulated is unknown ...

```
1.  // variable to be manipulated
2.  unsigned char my_variable = 0b10001100;
3.  // bitmask - definition of the bit positions to be manipulated
4.  unsigned char bit_mask = ((1 << 2) | (1 << 5));
5.  // Clearing bits by ANDING a variable with a negated bitmask ...
6.  my_variable = my_variable & ~bitmask;
```

Notes

- The previous expression in detail ...

```
1.  (1 << 2)   ⟶  (0b00000001 << 2)   ⟶   0b 0000 0100
2.  (1 << 5)   ⟶  (0b00000001 << 5)   ⟶   0b 0010 0000
3.                                        ---------------    // bitwise OR (bitmask)
4.                          bit_mask =    0b 0010 0100
5.                         ~bit_mask =    0b 1101 1011    // negate the bitmask
6.                       my_variable =    0b 1000 1100
7.                                        ---------------    // bitwise AND (variable)
8.                       my_variable =    0b 1000 1000
```

- Bits 2 and 5 are cleared, all other bits remain unchanged.

AND truth table

x	y	q
0	0	0
0	1	0
1	0	0
1	1	1

OR truth table

x	y	q
0	0	0
0	1	1
1	0	1
1	1	1

Clearing Individual Bits

```
1. my_variable = my_variable & ~bitmask;
```

Notes

- Please note that the previous expression can also be expressed using the following shorthand assignment operator.

```
1. my_variable &= ~bitmask;
```

Bit Manipulation in C – Inverting/Toggeling Individual Bits

Embedded Programming in C in a Nutshell

Inverting/Toggeling Individual Bits

```
1.  // variable to be manipulated
2.  unsigned char my_variable = 0b10001100;
3.  // bitmask - definition of the bit positions to be manipulated
4.  unsigned char bit_mask = ((1 << 2) | (1 << 5));
5.  // Inverting/Toggeling bits using the bitwise XOR operator
6.  my_variable = my_variable ^ bitmask;
```

XOR truth table

x	y	q
0	0	0
0	1	1
1	0	1
1	1	0

Notes

- The previous expression in detail ...

```
1.  (1 << 2)   ⟶  (0b00000001 << 2)   ⟶   0b 0000 0100
2.  (1 << 5)   ⟶  (0b00000001 << 5)   ⟶   0b 0010 0000
3.                                        ---------------    // bitwise OR (bitmask)
4.                              bit_mask = 0b 0010 0100
5.                           my_variable = 0b 1000 1100
6.                                        ---------------    // bitwise XOR (variable)
7.                           my_variable = 0b 1010 1000
```

- Bits 2 and 5 are toggled, all other bits remain unchanged.

Inverting/Toggeling Individual Bits

```
1. my_variable = my_variable ^ bitmask;
```

Notes

- Please note that the previous expression can also be expressed using the following shorthand assignment operator.

```
1. my_variable ^= bitmask;
```

Bit Manipulation in C – Macros

Embedded Programming in C in a Nutshell

Bit Manipulation in C – Macros

```
1.  #define set_bit(byte,nbit)   ((byte) |=  (1<<(nbit)))
2.  #define clr_bit(byte,nbit)   ((byte) &= ~(1<<(nbit)))
3.  #define inv_bit(byte,nbit)   ((byte) ^=  (1<<(nbit)))
4.
```

Notes

- Text substitution only, no data type checking

Bit Manipulation in C – Testing the State of Individual Bits

Embedded Programming in C in a Nutshell

Testing the State of Individual Bits

- In addition to changing individual or several bits as described in the previous sections, it is often necessary to determine the state of a bit.
- If you would like to check whether one or more bits in a target variable/register are set or not, you have to link this variable with a corresponding bitmask via a bitwise logical AND operation.
- The bitmask must have a ‘1’ at the position of the bit to be checked, but a '0' at all other positions. In this way, we are isolating the bit/bits of interest …

Testing the State of Individual Bits

normally, the state of the variable/register to be manipulated is unknown ...

```
1. // variable to be tested
2. unsigned char my_variable = 0b10001100;
3. // bit_state – result variable
4. unsigned char bit_state;
5. // 'reading' bit 2
6. bit_state = my_variable & (1 << 2);
```

AND truth table

x	y	q
0	0	0
0	1	0
1	0	0
1	1	1

Notes

- The previous expression in detail ...

```
1. (1 << 2)   ⟶   (0b00000001 << 2)   ⟶   0b 0000 0100
2.                                  my_variable = 0b 1000 1100
3.                                                ——————————    // bitwise AND (variable)
4.                                  my_variable = 0b 0000 0100
```

- Please be aware that the result of the previous operation is not equal to 1!
- If the bit tested is set, the result is equal to two to the power of the bit position tested!

Testing the State of Individual Bits

```
1. // test, if bit '2' is set
2. if(my_variable & (1 << 2))
3.     // bit '2'is set
4. else
5.     // bit '2'is not set …
```

Notes

- This expression can alternatively be written as follows

```
1. // test, if bit '2' is set
2. if((my_variable & (1 << 2)) == 0x04)
3.     // bit '2'is set
4. else
5.     // bit '2'is not set …
```

Testing the State of Individual Bits

```
1. // test, if bit '2' is not set
2. if(!(my_variable & (1 << 2)))
3.     // bit '2'is not set …
4. else
5.     // bit '2'is set …
```

Notes

- This expression can alternatively be written as follows

```
1. // test, if bit '2' is not set
2. if((my_variable & (1 << 2)) != 0x04)
3.     // bit '2'is not set …
4. else
5.     // bit '2'is set …
```

Testing the State of Individual Bits

- In the following, the state of several bits (bits 2 & 5) in an 8-bit wide target variable is to be determined as an example.

Testing the State of Individual Bits

```
1. // test if two bits are set
2. if((my_variable & ((1 << 2) | (1 << 5))) == 0x24 )
3.     // bit '2' and bit bit '5' are set …
4. else
5.     // …
```

```
1. // test if at least one bit is set
2. if((my_variable & ((1 << 2) | (1 << 5))) != 0 )
3.     // at least one bit set …
4. else
5.     // …
```

```
1. // test if both bits are not set
2. if((my_variable & ((1 << 2) | (1 << 5))) == 0 )
3.     // both bits of interest are not set
4. else
5.     // …
```

Bit Manipulation in C – Read/Get the State of Individual Bits

Embedded Programming in C in a Nutshell

Read/Get the State of Individual Bits

Notes

- Sometimes we want the exact state of a dedicated bit in a variable/register as a values.
- This can be accomplished with the ternary operator (remember SystemVerilog):

```
1. bit_state =(my_variable & (1 << X)) ? 1 : 0;
```

- If bit `X` in `my_variable` is 1, `bit_state` will be one, else `bit_state` will be zero.

- The following code snippet provides an alternative implementation:

```
1. bit_state =(my_variable >> X) & 1;
```

- Here, the particular bit of interest is shift to the LSB location. It is then ANDed with one. If this bit is zero, the result of the operation is zero, otherwise it is one.

Bit Manipulation in C – Waiting for Bit State Changes

Embedded Programming in C in a Nutshell

Waiting for Bit State Changes

- Another important application in this context is waiting for a certain bit state.
- For example, one wants to wait until a certain bit in a register changes its state (from '0' to '1' or vice versa).
- Thus one checks permanently the value of a bit and reacts in case of a change.
- This function is implemented, as described on the following slides, on the basis of a simple while loop construct.
- If the microcontroller waits for an event by actively sampling the status of a bit (or even more ...) without doing something in between, this procedure is called **busy waiting** or **polling**.

Waiting for Bit State Changes

```
1. // wait until bit '2' is set, do nothing until then
2. while(!(REGISTER & ((1 << 2))) {   };
3.
```

```
1. // wait until bit '2' is cleared, do nothing until then
2. while((REGISTER & ((1 << 2))) {   };
3.
```

Notes

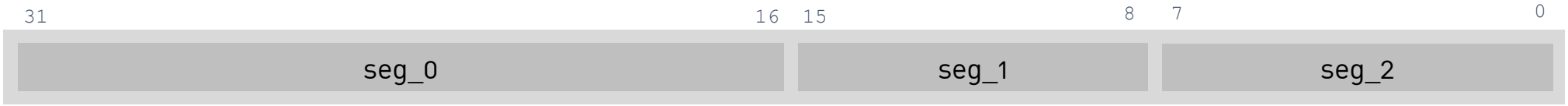
- Please be aware that in both examples **the loop body is empty**! No instructions will be executed by the CPU. So we are just actively waiting without doing something else …
- The while loop is executed as long as the loop condition is met. In the first case, REGISTER is linked to the bit mask (0b000010) via a bitwise logical AND operation.
- If bit '2' is set in this case, the result of this operation is identical to the value of the bitmask. Otherwise the result is zero.
- Depending on whether this expression is used directly or inverted in the loop check, one thus waits actively for deletion or setting of bit '2'.

Packing/Unpacking of Data Sections

Embedded Programming in C in a Nutshell

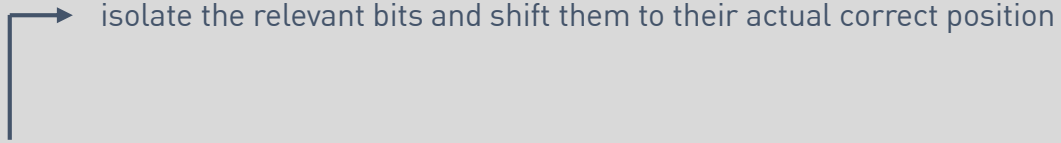
Packing/Unpacking of Data Sections

- Memory mapped I/O registers often contain multiple different fields, which are extracted and separated after reading the complete register content.
- The following example assumes that a 32-bit wide register is subdivided into a single 16-bit part as well as two 8-bit parts.



Unpacking of Data Sections

```
1.  alt_u32 data = read_reg(REG_ADDRESS);
2.
3.  alt_u8  seg_1, seg_2;
4.  alt_u16 seg_0;
5.
6.  seg_0 = (alt_u16)((data & 0xFFFF0000) >> 16);
7.  seg_1 = (alt_u8) ((data & 0x0000FF00) >>  8);
8.  seg_2 = (alt_u8) ((data & 0x000000FF));
9.
```



Notes

- Unpacking/Extracting ...

Packing of Data Sections

```
1.   alt_u32 data_reg;
2.
3.   alt_u8  seg_1, seg_2;
4.   alt_u16 seg_0;
5.
6.  seg_0 = ...
7.  seg_1 = ...
8.  seg_2 = ...
9.
10. data =  (alt_u32) seg_0;
11. data = ((alt_u32) seg_1) | (data << 8);
12. data = ((alt_u32) seg_2) | (data << 8);
13.
14. write_reg(REG_ADDRESS,data);
```

Notes

- Packing bits …