Special Topics for Embedded Programming

Fall 2019

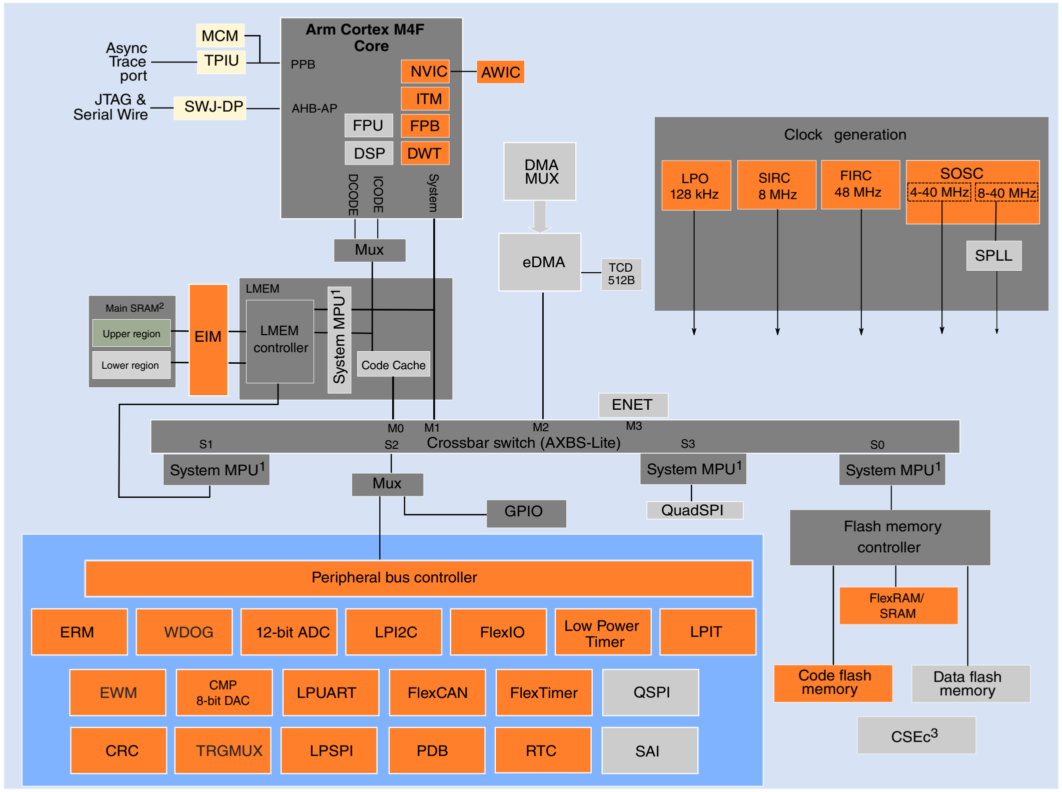
Reference: The C Programming Language by Kernighan & Ritchie

Overview of Topics

- Microprocessor architecture
  - Peripherals
  - Registers
  - Memory mapped I/O
- C programming for embedded systems
- Lab 1: General Purpose I/O
  - Read data from input pins and write to output pins on the S32K144
  - GPIO example code

S32K144: 32-bit ARM Cortex-M4F MCU

- 32-bit ARM Cortex-M4F MCU
- Up to 112 MHz (HSRUN mode) or 80 MHz (RUN mode)
- Ambient temperature range:
  -40° C to 105/125° C
- Modified Harvard Architecture with up to 2 MB code memory and 256 KB SRAM
- 89 pin GPIO individually programmable as input, output or special function
- 3 Controller Area Network (FlexCAN) units, one with CAN FD
- Two 12-bit, 16-channel analog-to-digital converters (ADCs)
- Four 8-channel FlexTimer modules with 16-bit timer
- One 4-channel Low Power Interrupt Timer (LPIT) with 16-bit counter



Registers and Addresses

- Microprocessor memory has **location** (an **address**, specified as a hexadecimal number) and contents (the data stored at a specific address in memory)
- **Registers** are memory locations used for calculations, to initialize the processor or check its status, or to access peripheral devices.
  - General purpose registers
  - Special purpose registers
- Memory-mapped I/O refers to communication between the CPU and its peripheral devices using the same instructions, and same bus, as between CPU and memory
  - Peripheral devices are accessed by writing to and reading from special purpose registers
  - Address maps in the user manual define the address, name and size of each special purpose register

General Purpose Registers

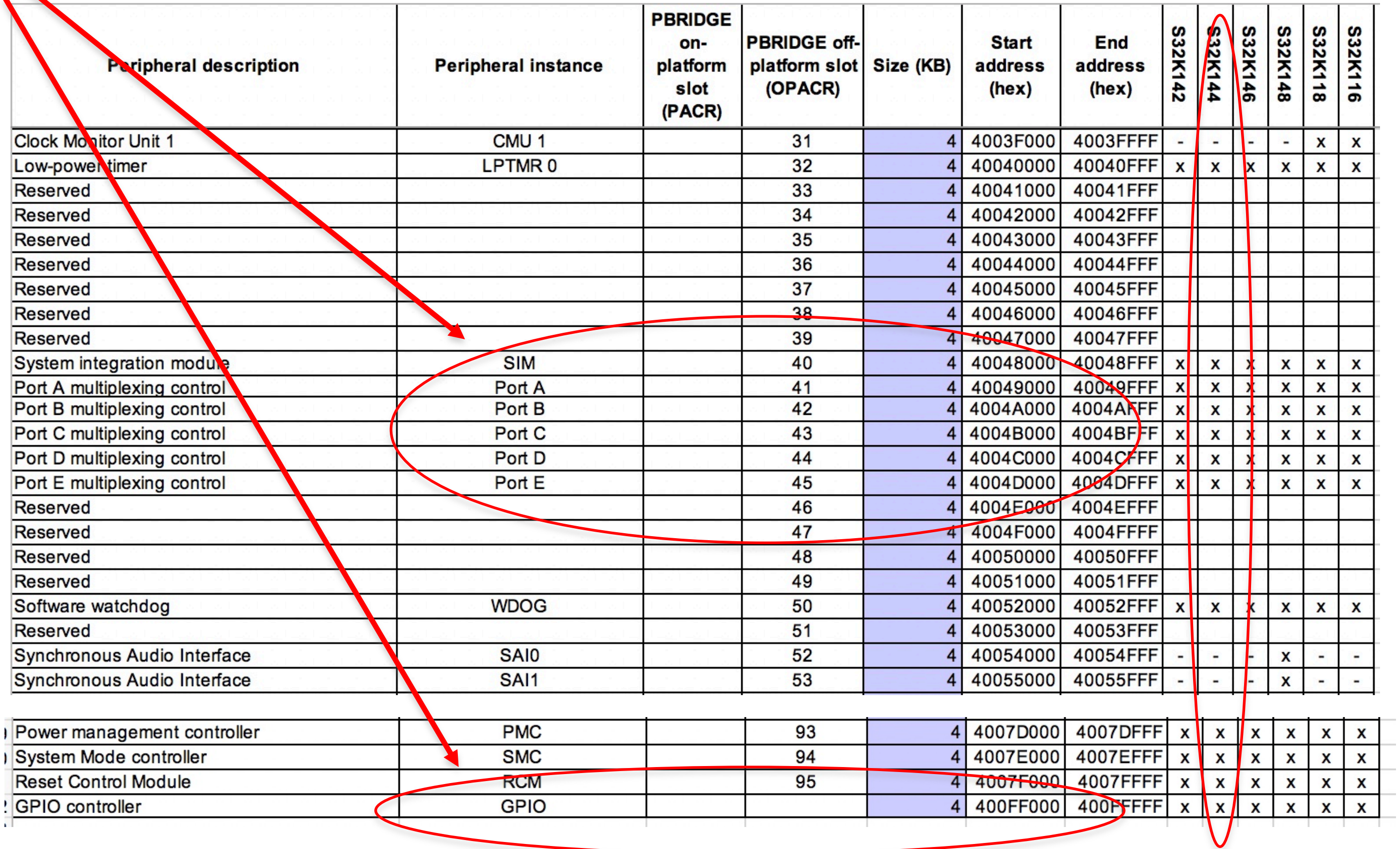
- Generally used for storage and manipulation of data required in the program execution
- Faster access than main memory
- S32K144 has thirteen 32-bit general purpose registers for data operations
- C compilers automatically use these registers as resources to load/store data and perform calculations
  - Unless you're writing assembly code, you won't have to read from or write to any GPR
  - We'll discuss assembly code, but we won't write any

Special Purpose Registers: Memory Mapped I/O

- Access peripherals by writing to and reading from special purpose registers
- Each peripheral has a fixed range of memory assigned to it
- Lots of registers associated with each peripheral
  - Peripheral configuration registers
  - Peripheral status registers
  - inputs from the hardware
  - outputs to the hardware

Memory Map

- See the spreadsheet S32K1xx_Memory_Map.xlsx
- Base address of all memory modules and peripheral devices
- Subsequent references to register locations are as offsets from the base address (start address) given in the memory map.
- Will need some of these in Lab 1



Peripheral description	Peripheral instance	PBRIDGE on-platform slot (PACR)	PBRIDGE off-platform slot (OPACR)	Size (KB)	Start address (hex)	End address (hex)	S32K142	S32K144	S32K146	S32K148	S32K118	S32K116
Clock Monitor Unit 1	CMU 1		31	4	4003F000	4003FFFF	-	-	-	-	x	x
Low-power timer	LPTMR 0		32	4	40040000	40040FFF	x	x	x	x	x	x
Reserved			33	4	40041000	40041FFF						
Reserved			34	4	40042000	40042FFF						
Reserved			35	4	40043000	40043FFF						
Reserved			36	4	40044000	40044FFF						
Reserved			37	4	40045000	40045FFF						
Reserved			38	4	40046000	40046FFF						
Reserved			39	4	40047000	40047FFF						
System integration module	SIM		40	4	40048000	40048FFF	x	x	x	x	x	x
Port A multiplexing control	Port A		41	4	40049000	40049FFF	x	x	x	x	x	x
Port B multiplexing control	Port B		42	4	4004A000	4004AFFF	x	x	x	x	x	x
Port C multiplexing control	Port C		43	4	4004B000	4004BFFF	x	x	x	x	x	x
Port D multiplexing control	Port D		44	4	4004C000	4004CFFF	x	x	x	x	x	x
Port E multiplexing control	Port E		45	4	4004D000	4004DFFF	x	x	x	x	x	x
Reserved			46	4	4004E000	4004EFFF						
Reserved			47	4	4004F000	4004FFFF						
Reserved			48	4	40050000	40050FFF						
Reserved			49	4	40051000	40051FFF						
Software watchdog	WDOG		50	4	40052000	40052FFF	x	x	x	x	x	x
Reserved			51	4	40053000	40053FFF						
Synchronous Audio Interface	SAI0		52	4	40054000	40054FFF	-	-	-	x	-	-
Synchronous Audio Interface	SAI1		53	4	40055000	40055FFF	-	-	-	x	-	-
Power management controller	PMC		93	4	4007D000	4007DFFF	x	x	x	x	x	x
System Mode controller	SMC		94	4	4007E000	4007EFFF	x	x	x	x	x	x
Reset Control Module	RCM		95	4	4007F000	4007FFFF	x	x	x	x	x	x
GPIO controller	GPIO			4	400FF000	400FFFFF	x	x	x	x	x	x

Memory Addresses

- Each memory address refers to an 8-bit *byte* of memory: the smallest unit of memory that is addressable.
- Memory addresses are *hexadecimal.*
- Binary (base 2)
  - Binary number 1011 converted to decimal is: $(1 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) = 11$
- Hexadecimal (base 16)
  - 0–9 represent values zero to nine
  - A, B, C, D, E, F represent values ten to fifteen
  - Hexadecimal number 2AF3 converted to decimal is:
  - $(3 \times 16^0) + (15 \times 16^1) + (10 \times 16^2) + (2 \times 16^3) = 10{,}995$
  - Compiler recognizes 0x2af3 or 0x2AF3 as a hexadecimal number
- Hex to binary conversion (and back) is easy: Each hex digit is a 4-bit “nibble” (2 nibbles are an 8-bit byte)
  - 2AF3 converted to binary is: 0010 1010 1111 0011

C Programming for Embedded Systems

Primitive Data Types, Data Declaration

- Integer data types
  - Have both size and sign
  - char (8-bit)
  - short (16-bit)
  - int (32-bit)
  - long (32-bit)
  - signed (positive and negative)
  - unsigned (positive only)
- Floating-point types
  - Only have size
  - Can always be positive or negative
  - float (32-bit)
  - double (64-bit)
  - 12.34: constant of type double
  - 12.34f or 12.34F: constant of type float
- Data declarations should be at the top of a code block

```
{
/* Top of Code Block
*/
signed char A;
char input;
unsigned short var;
int output;
unsigned long var2;

float realNum;
double realNum2;
…
}
```

Data Types Defined in `stdint.h`

- Standard C data types
  - `typedef char int8_t;`
  - `typedef short int16_t;`
  - `typedef long int32_t;`

  - `typedef unsigned char uint8_t;`
  - `typedef unsigned short uint16_t;`
  - `typedef unsigned long uint32_t;`

Functions and Function Prototypes

- Function Prototype declares name, parameters and return type prior to the function’s actual declaration
- Information for the compiler; does not include the actual function code
- You will be given function prototypes to access peripherals
- Pro forma: `RetType FcnName (ArgType ArgName, … );`

```
int Sum (int a, int b);    /* Function Prototype */
void main()
{
       c = Sum ( 2, 5 );        /* Function call */
}
       int Sum (int a, int b)   /* Function Definition */
              {
           return ( a + b );
              }
```

C vs. C++

- C++ language features cannot be used
  - No `new, delete, class`
- Variables must be declared at top of code blocks

```
{
int j;
double x;

/* code */
int q;                /* only in C++, not in C */
/* code */
}
```

Useful Features for Embedded Code

- Storage Class Speciers & Type Qualifers
  - volatile
  - const
  - static
- Pointers
- Structures
- Bit Operations
- Integer Conversions

Volatile Type Qualifier

- Variables that are reused repeatedly in different parts of the code are often identified by compilers for optimization.
  - These variables are often stored in an internal register that is read from or written to whenever the variable is accessed in the code.
  - This optimizes performance and can be a useful feature
- Problem for embedded code: Some memory values may change without software action!
  - Example: Consider a memory-mapped register representing a DIP-switch input
  - Register is read and saved into a general-purpose register
  - Program will keep reading the same value, even if hardware has changed
- Use `volatile` qualifier:
  - Value is loaded from or stored to memory every time it is referenced
  - Example: `volatile unsigned char var_name;`

Const Type Qualifier

- The type qualifier `const` is used to declare that a variable is read-only and may not be changed in software.
  - Example: `uint16_t const max_temp = 1000;`
  - Could also write: `#define max_temp 1000`

- If a variable of type `const` is stored in a memory-mapped register, then its value cannot be changed in software BUT could be changed externally. In this case the variable should be defined as both `volatile` and `const`
  - Example: `volatile const uint32_t var_name;`

- NXP shorthand for `volatile` and `volatile const:`

```
– #define    __I   volatile const  /*'read only' */
– #define    __O   volatile       /*'write only' */
– #define    __IO  volatile       /*'read/write’ */
```

Static Storage Class

- Two uses: in a file and in a code block (e.g., function)
    1) Declaring a variable `static` in a file limits the scope of that variable to the file in which it is declared. The variable will not conflict with other variables of the same name defined in other files. Similarly, declaring a function static in a file limits the scope of that function to the file.
    2) Declaring a variable `static` in a function allocates it a persistent memory location and it retains its value between successive function calls.
- In embedded code we often want a variable to retain its value between function calls
    - Consider a function that senses a change in the crankshaft angle: The function needs to know the previous angle in order to compute the difference
- Example: `static int x = 2 ;`

Global Variables

- Variables that are visible to routines in *all* files, not just the file in which they are defined
  - such variables are said to have *program scope*
- Use of global variables can make a program difficult to understand and maintain
- To make a variable local, declare it with the `static` keyword:

```
int j;              /*  global   */
static int k;       /*  local     */
main ()
{
   …
     }
```

Pointers

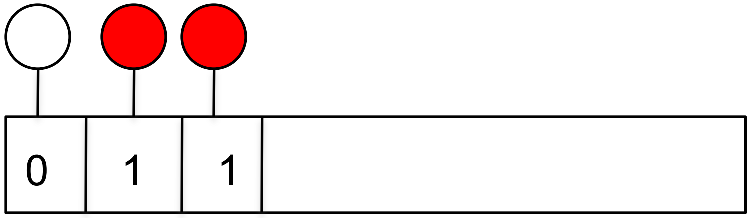
- Every variable has an *address* in memory and a *value*
- A pointer is a variable that stores an address
  - The value of a pointer is the location of another variable
- The size of a pointer variable is the size of an address
  - 4 bytes (32 bits) for the S32K144
- Two operators used with pointers
  - `&` operator returns the address of a variable
  - `*` is used to "de-reference" a pointer (*i.e.*, to access the *value* at the *address* stored by a pointer

Simple Pointer Example

Address	Value	Variable
0x100	5	x
0x104	5	y
0x108	0x100	ptr

```
    int x, y;                /* x is located at address 0x100 */
    int *ptr;               /* This is how to declare a pointer. Pointer
                             /* ptr points to an int */
    x = 5;
    ptr = &x;               /* The value of ptr is now 0x100 */
    y = *ptr;               /* y now has the value 5 */
    *ptr = 6;               /* The value at address 0x100 is 6 */
```

Another Way to Assign Pointers



- Declare a pointer, `p`, that points to a `uint32_t`

```
volatile uint32_t      *p;
```

- Allocate the memory and assign the address of the I/O memory location to the pointer ("dereference `p`")

```
p = (volatile uint32_t *) 0x30610000;
/* 4-byte long, address of some memory-
mapped register */
```

- Set the contents of memory location 0x30610000,

```
*p = 0x7FFFFFFF;   /* Turn on all but
the leftmost bit */
```

Pointer Arithmetic

- Recall that each memory address refers to an 8-bit byte of memory.
- Suppose that `p` is a pointer to a certain type of object (e.g., int, short, char). Then `p+i` points to the i'th object after the one `p` points to.
- Example

```
    int *p, *p1, *p3
    p = (int *) 0x1000;
    p1 = p++;  /* value of p1 is 0x1004  */
    p3 = p+3;  /* value of p3 is 0x100C  */
```

- Pointer indexing

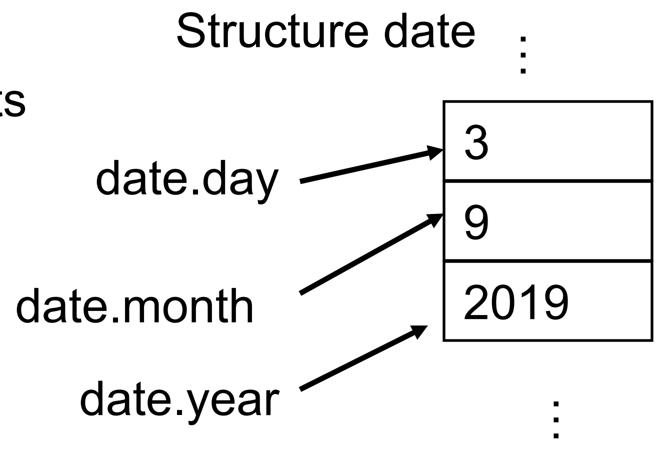
```
    p[i] = *(p+i)
```

- Application: memory-mapped registers used to configure HW peripherals. Define pointer to first register and use indexing to access the rest

Structures

- A `struct` holds multiple variables
  - Divides range of memory into pieces that can be referenced individually
  - Example: The date consists of several parts

```
 struct date {
 int day;
 int month;
 int year;
};
```



- Recall: We can treat a peripheral's memory map as a range of memory.
- Result: We can use a structure and its member variables to access peripheral registers.

Structures

- Access structure member variables using "." and "->"
  - . is used with structures
  - -> is used with pointers-to-structures
- Example
  - `current_time` is a variable of type `time`
  - `pcurrent_time` is a pointer to a structure (pointers to structures have special properties which will be useful in accessing arrays of structures)
  - Assign `hour, minute` and `second`
  - Increment `minute`

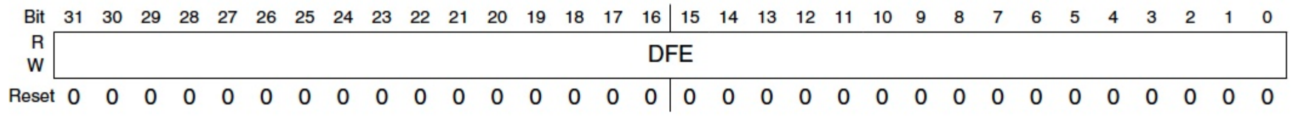
```
/* Definition */
struct time {
           int hour, minute,
second;
         };
Void main()
{
  struct time current_time;
  struct time *pcurrent_time;

  current_time.hour =  12;
  current_time.minute = 0;
  current_time.second = 0;

  pcurrent_time = &current_time;
  pcurrent_time -> minute++;
};
  /* same as
(*pcurrent_time).minute++ */
```

Constants and Bit Operations

Tools for Manipulating Values

Integer Constants

- Binary constant: 0b*number*
  - Example: `0b10110000`
- Hexadecimal constant: 0x*number*
  - Example: `0xffff` or `0xFFFF`
- Note: in the S32K manual binary, decimal, and hex numbers are often indicated with a suffix: `b`, `d`, or `h`.
  - Example: `14d = 1110b = Eh`
  - The "`d`" is often omitted
- Unsigned constants: to specify that a numerical value is to be stored as an unsigned data type, use suffix `u`
  - Example: `1234` is stored as a (signed) `int`
  - `1234u` is stored as an `unsigned int`
- In the S32K reference manual bits are numbered right to left starting at 0:



Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	DFE																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Logical Bit Operations

- Several operations work directly on the bits of a binary number, rather than its numerical value. Let `a` and `b` be n-bit binary numbers.

- Bitwise AND (`a&b`): if both bits are 1, set result to 1. Otherwise set result to 0.
  - `0b1001&0b1010 = 0b1000`
- Bitwise OR (`a|b`): if both bits are 0, set result to 0. Otherwise, set result to 1.
  - `0b1001|0b1100 = 0b1101`
- Bitwise XOR (`a^b`) (exclusive OR): if exactly one bit is 1, set result to 1. Otherwise set result to 0.
  - `0b1001^0b1100 = 0b0101`
- One’s complement (`~a`): set 1 bits to 0 and 0 bits to 1
  - `~0b1001 = 0b0110`

Logical Bit Shifting

Let `a` be a binary number stored in an `n`-bit register, and let `x` be a nonnegative integer.

- Left shift (`a<<x`): move all bits `x` places to the left.

```
0b0011 << 2 = 0b1100
```

- Right shift (`a>>x`): : move all bits `x` places to the right.

```
0b1001 >> 2 = 0b0010
```

- In either case, bits shifted out of the register are discarded, and vacancies created are filled with zeros.
- Left shift `x` is equivalent to multiplication by 2^x if there is no overflow or multiplication by 2^x (modulo 2^n) if overflow occurs.

$3 = 0b0011 << 2 = 0b1100 = 3*2^2 = 12$

$10 = 0b1010 << 2 = 0b101000 = 10*2^2 = 40$

$10 = 0b1010 << 2 = 0b1000 = 40 (\mathrm{mod}\ 2^4) = 8$

- Right shift `x` is equivalent to division by 2^x (rounded down to integer)

$12 = 0b1100 >> 2 = 0b0011 = 3$

$10 = 0b1010 >> 2 = 0b0010 = \lfloor 2.5 \rfloor = 2$

Masking, Setting, and Clearing Bits

- Masking – bitwise AND may be used to extract bits from a number
  - Example: extract bits 5 and 6 from an 8 bit value and right justify the result.

    `(x >> 5) & 0b011` or `(x & 0x30) >> 5`

- Set bits – bitwise OR may be used to set bits to 1
  - Example: set bit 5 of `x` to 1

    `x = x | (1 << 5)` or `x |= (1 << 5)`

- Clear bits – bitwise OR may be used to clear bits to 0
  - Example: clear bits 5 and 6 of `x` to 0

    `x = x & ~(0b11 << 5)` or `x &= ~(0b11 << 5)`

- Note: when setting and clearing bits, the remaining bits are unchanged

Type Conversions

- Explicit Casts
  - For specifying the new data type
  - Syntax: `(type-name)expression`
  - Example: `(int)largeVar`
- Integral Promotion
  - Before basic operation (`+ - * /`), both operands converted to same type
  - The smaller type is "promoted" (increased in size) to the larger type
  - Value of promoted type is preserved
- Implicit Casts
  - Assigning a value into a different type
  - Widening conversion – preserve value of expression

```
short x = 10;  long y = x;
```

  - Narrowing conversions – do NOT preserve value

```
unsigned long x = 257;
unsigned char y = x;
```

Integer Division

- When dividing two numbers, may receive unexpected results
- If both operands are integers, then result is integer
  - Expected result of division is truncated to fit into an integer
- If one operand is floating-point, then result is floating-point
  - Integer operand is promoted to floating-point
  - Receive expected result

```
/* floating-point results */
(5.0 / 2.0) →  2.5  /* no promotion, result is float */
(5.0 / 2)  →  2.5   /* operand 2 promoted to float */
(5 / 2.0)  →  2.5   /* operand 5 promoted to float */

/* integer-valued results */
(5 / 2)  →  2 /* no promotion, result is integer */
```