EE443 - Embedded Systems

Lecture 5
Review of C Data Types

Contents:

5.1 Predefined Data Types
  5.1.1 Local and Global Variables
  5.1.2 Static Attribute
5.2 Pointers in C
  5.2.1 Using Pointers
5.3 Structured Data Types
  5.4.1 Unions
5.4 Type Casting
5.5 Notes to the Programmer
5.6 Compatibility Matters

5.1 Predefined Data Types

We are required to declare all types of variables we use in a C program. This requirement may sound like an unnecessary detail for the beginners who have limited programming experience in other languages. On the other hand, the same requirement brings in some advantages for those working on projects for professional purposes:

1. If somebody else reads your program, then he/she will have a better idea about the data types in your code. This will make it easier to understand the program.

2. In case you make a typo (typing mistake) or some other syntax error, it will be easier for the compiler to catch your mistake. You are likely to get a more meaningful error message, and locate your mistake easily. On the other hand, if the compiler ignores your typo and automatically gives you another variable, then you may end up spending hours trying to find that new variable you are not aware of.

3. The limitations of data operations are well-defined when the data types are known. You can figure out when an overflow condition may occur in your program regardless of the programming platform or the microprocessor hardware you are using.

The standard numeric data types in C are listed below. The C language keywords are written in blue and the optional entries are enclosed in square brackets, "[...]".

```
[signed, unsigned] char             // single-byte integer
[signed, unsigned] [short] [int]    // two-byte integer
[signed, unsigned] [long] [int]     // four-byte integer
float       // four-byte single precision floating point
double      // eight-byte double precision floating point
```

Following examples give the formal declarations for integer data types and the corresponding numeric range:

```
signed char         MySbyte;  // -128..+127
unsigned char       MyUbyte;  // 0..+255
signed short int    MySSint;  // -32,768..+32,767
unsigned short int  MyUSint;  // 0..+65,535
signed long int     MySLint;  // -2,147,483,648..+2,147,483,647
unsigned long int   MyULint;  // 0..+4,294,967,295
```

C compilers accept declaration of integer data types omitting one or more of the keywords. We often try to minimize memory usage and maximize processing speed in programming projects for embedded systems. For this reason, we should use the formal declarations to choose the optimum data type that serves our purpose.

Standard C language has no specific data type to store strings or data in basic text format. Character strings are stored in simple arrays of `char` data type. The standard C library functions described in **<string.h>** definition file provide the necessary set of string processing tools (e.g. comparison, search, concatenation, etc.). All of these string processing functions operate on arrays of characters.

5.1.1 Local and Global Variables

Programs may contain several functions and service routines to supplement a main program. Variables have different ***scope*** of accessibility depending on where they are declared in a source file. If a variable is ***in scope*** in a certain part of a program then it can be read and altered in that part only. The variables that are declared or defined as a parameter in a function are ***local variables***. Local variables are defined in the scope of a single function, and they are not accessible in other functions. Variables that are declared outside of all functions are ***global variables***. Global variables are accessible in all functions after they are declared in a source file.

In the following example, `Counter` is a global variable that is accessed in the **main** procedure and in the function `UpdateCounter`.

```
signed short int  Counter = 0;   // declare global variable

int  main()
{ // Local variables of main function:
  signed short int  Limit;
  unsigned char     Abyte, Bbyte;
  . . .
  if (Counter == Limit)
  { Abyte = 0;
    . . .
}

void  UpdateCounter(signed short int Increment)
{ // Local variables of UpdateCounter function:
  unsigned char     Ain;
  . . .
  if (Ain == 0x41)
    Counter += Increment;
  . . .
}
```

Note that, if a function declares a local variable with the same name given to a global variable then a second variable is created. The main program given below declares `Counter` as a local variable. All access to `Counter` in the main program involves the local variable only, and it is independent of the global `Counter`.

```
signed short int  Counter = 0;   // declare global variable

int  main()
{ // Local variables of main function:
  signed short int  Counter;
  signed short int  Limit;
  unsigned char     Abyte, Bbyte;
  . . .
  if (Counter == Limit)
  { Abyte = 0;
    . . .
}
```

Group of functions that perform the tasks related to a certain purpose or target hardware are usually written as library modules in separate source files. Dividing a big programming project into several source files has several advantages compared to keeping all of the source code in a single long file :

1. Shorter source files can be edited more efficiently without wasting time scrolling through a single big file.

2. Variables and other critical data related to a specific group of tasks can be isolated from the other parts of the program.

3. When there are several engineers working on a project, each engineer can work on a separate module.

The C compiler processes the source files of a project seperately and it creates an object code file corresponding to each source file. In the next compilation step, a ***linker*** utility makes the necessary connections between all object codes and creates a single executable code for the project.

Global variables declared in a source file can be made accessible in the other files of a project by using the `extern` attribute. In the following example, `main` procedure and `UpdateCounter` are written in separate files. `Counter` is declared as a global variable in the file `CounterSource.c`. Specification of `Counter` with the `extern` attribute in `MainSource.c` does not create a global variable. It just tells the compiler that `Counter` variable was declared in another source file of the project. Similarly, the prototype declaration of `UpdateCounter` specifies the parameters of this function and what it returns at the end. All of this information is used to establish the necessary links between the object codes when the linker creates the executable code.

```
// Source file: MainSource.c
// Specify Counter as an external variable declared in another source
file:
extern signed short int  Counter;
// Prototype of UpdateCounter function defined in another source file:
void  UpdateCounter(signed short int Increment);

int  main()
{ // Local variables of main function:
  signed short int  Limit;
  unsigned char     Abyte, Bbyte;
  . . .
  if (Counter == Limit)
  { Abyte = 0;
    . . .
}

// Source file: CounterSource.c
signed short int  Counter = 0;   // declare global variable
void  UpdateCounter(signed short int Increment)
{ // Local variables of UpdateCounter function:
  unsigned char     Ain;
  . . .
  if (Ain == 0x41)
    Counter += Increment;
  . . .
}
```

Compilers with optimization utilities check if variables are really necessary or not. If the global variable `Counter` does not affect any other variable or output value in `CounterSource.c`, then the optimization utility eliminates all statements related to `Counter`, assuming that it is a useless variable. The optimization utility does not know the fact that `Counter` was used in `MainSource.c` since the compiler focuses on only one source file at a time. If `Counter` is eliminated from `CounterSource.c`, then the linker will give an error message stating that `Counter` is not declared in the project. The `volatile` attribute used in global declarations prevents elimination of a variable in such cases:

```
// Source file: CounterSource.c
volatile signed short int  Counter = 0;   // declare global variable
. . .
```

Global variables increase the risk of errors in a program and they should be used only when they are absolutely necessary. If something goes wrong with a global variable, then debugging of the problem requires verification of all functions in all source files that can access that variable. Experienced programmers write modular programs where scope of variables are limited to small modules and data space of each module is isolated from others as much as possible

5.1.2 Static Attribute

Local variables require memory during execution of a single function. The memory reserved for local variables becomes disposable after the function returns, and the compiler may use the same memory space for local variables declared in other functions. As a result, data stored in a local variable may change randomly between the function calls. On the other hand, if a local variable is declared with the `static` attribute, then it has its dedicated memory and its value cannot be changed by other functions.

The counter operations in the previous example can be implemented without using `Counter` as a global variable. Changing `Counter` into a local variable has three requirements: **1)** main program should still be able to read the `Counter` value, **2)** `Counter` value should not change, **3)** `Counter` should be initialized properly. The first requirement is satisfied by using a modified `UpdateCounter` function that returns the last `Counter` value. The other requirements are satisfied by declaring `Counter` as a local variable with the `static` attribute.

```
// Source file: MainSource.c
// Prototype of UpdateCounter function defined in another source file:
signed short int  UpdateCounter(signed short int Increment);

int  main()
{ // Local variables of main function:
  signed short int  Limit;
  unsigned char     Abyte, Bbyte;
  . . .
  if (UpdateCounter(0) == Limit)
  { Abyte = 0;
    . . .
}

// Source file: CounterSource.c
signed short int  UpdateCounter(signed short int Increment)
{ // Local variables of UpdateCounter function:
  static signed short int  Counter = 0;
  unsigned char     Ain;
  . . .
  if (Ain == 0x41)
    Counter += Increment;
  . . .
  return Counter;
}
```

Declaration of a local variable with the `static` attribute has two consequences:

1. The local variable keeps its value after the function returns. In the last example, next time `UpdateCounter` function is called, `Counter` will still have the value returned in the previous function call.

2. If an initial value is specified in declaration of a static variable, then initialization is done only once before the first function call. In a typical microprocessor, initialization of static local variables is performed after a reset condition. Normal local variables without the `static` attribute are initialized every time the function is called.

5.2 Pointers in C

Pointers are stored in memory and accessed through the symbols we define just like any other variable. The numeric value stored in a pointer serves as an address, and the C compiler gives us the necessary means to access data utilizing this address information. Pointers are declared placing an asterisk (*****) before the variable name:

```
signed char         *SbytePtr;  // pointer to signed char
unsigned short int  *USintPtr;  // pointer to short int
signed long int     *LintPtr;   // pointer to long int
```

Each pointer stores the address of the first byte of the sequential memory locations reserved for a particular data type. The numeric variables declared in the previous section use up different number of bytes depending on the data type. `MySbyte` requires one byte, `MyUSint` requires two bytes and `MySLint` requires four bytes. All of the pointers declared above require the same amount of memory that depends on the computer system or the microprocessor we are programming. A typical microprocessor with 16-bit address bus uses two bytes of memory to store the address in a pointer. Similarly, a C program compiled for another system with 32-bit addressing would require four bytes of memory for each pointer.

All pointers use the same amount of physical memory regardless of the data type they are declared for. We are still required to specify the data type in pointer declarations because the compiler determines the operations in executable code according to the data type for a pointer. In other words, this is the way we tell the compiler what to do with the data accessed through a pointer as we will see in the following sections.

5.2.1 Using Pointers

The asterisk (*) character we used in pointer declarations has another meaning when it is used with pointers in executable statements. An asterisk placed before a pointer indicates that we want to access the memory location addressed by that pointer. Without that asterisk, we access the address information stored in the pointer itself just as we read or write another numeric variable. The following examples will clarify the usage of asterisk on the left hand side of assignments:

1. Write to the pointer itself, changing the address stored by the pointer: The `<expression>` must produce an **address** compatible with the pointer data type:

```
MyPointer = <expression>;
```

2. Write to the memory location addressed by the pointer: The `<expression>` must result in a **value** compatible with the data type addressed by the pointer:

```
*MyPointer = <expression>;
```

Similarly, we can read the pointer itself or the data addressed by a pointer on the right hand side of an assignment:

1. Read the address stored by the pointer: The `<target>` must be an address variable (i.e. another pointer) compatible with the **pointer** data type:

```
<target> = MyPointer;
```

2. Read the data at the memory location addressed by the pointer: The `<target>` must be compatible with the **data type** addressed by the pointer:

```
<target> = *MyPointer;
```

The asterisk character used before a pointer in the left or right hand side of executable statements means **"*access through*"** the address stored in that pointer. The ampersand (&) character allows us to obtain the address of a variable in a C program, reversing the function of "*" in a sense. It simply means **"*address of*"** when it precedes a symbol defined in the program. Consider the group of declarations given below:

```
unsigned char   Byte1, Byte2;  // two 1-byte integers
unsigned char  *BytePtr;       // pointer to unsigned char
```

The following three statements,

```
BytePtr = &Byte1;  // copy address of Byte1 into BytePtr
*BytePtr = 99;     // 99 goes to Byte1, through BytePtr
Byte2 = *BytePtr;  // Byte1 is copied to Byte2, through BytePtr
```

produce the same result that will be obtained with the statements,

```
Byte1 = 99;     // 99 goes to Byte1
Byte2 = Byte1;  // Byte1 is copied to Byte2
```

Note that the ampersand (**&**) character cannot be used on the left hand side of an assignment in the executable code. The address of a variable is determined by the compiler or the linker, and we cannot simply relocate a variable by assigning a new address to it during program execution.

Remember the short hand notations for incrementing and decrementing:

```
MyNumber ++;  // increment MyNumber by 1
MyNumber --;  // decrement MyNumber by 1
```

or for addition and subtraction:

```
MyNumber += <expression>;  // add result of <expression> to MyNumber
MyNumber -= <expression>;  // subtract result of <expression> from
MyNumber
```

Simple addition and subtraction operations on pointers are treated in a different way in C: The incremental addressing unit for a pointer is not one byte. Instead, the unit is given by the size of the data type of the pointer. If a pointer is used for addressing `long int`, then incrementing that pointer by **1** advances the address by **4** bytes. In the following examples, the comment fields give the resultant addresses assuming that the initial value of each pointer is `0x1A00` before the operation.

```
signed char          *BytePtr;   // pointer to signed char
unsigned short int   *ShortPtr;  // pointer to short int
signed long int      *LongPtr;   // pointer to long int
double               *DblPtr;    // pointer to double
```

```
// Initial value of each pointer is 0x1A00.
BytePtr += 2;  // incremental unit is 1 byte, new value=0x1A02
ShortPtr --;   // incremental unit is 2 bytes, new value=0x19FE
LongPtr ++;    // incremental unit is 4 bytes, new value=0x1A04
DblPtr += 2;   // incremental unit is 8 bytes, new value=0x1A10
```

An array name is a constant of pointer type when it is referenced without an index. In the following example, a buffer pointer is initialized to the starting address of memory reserved for an array .

```
signed short int   BufferArray[256];  // array memory reserved as buffer
signed short int  *BufferPtr;         // buffer pointer
. . .;
BufferPtr = BufferArray;  // initialize buffer pointer
```

The same initialization can also be done in the declaration of `BufferPtr`:

```
signed short int   BufferArray[256];  // array memory reserved as buffer
signed short int  *BufferPtr = BufferArray;  // buffer pointer
```

The initialization statements given above are equivalent to assigning `BufferPtr` to the address of the first array element by using ampersand (**&**) operator.

```
BufferPtr = &BufferArray[0];  // initialize buffer pointer
```

5.3 Structured Data Types

A structure defines a new data type that groups several predefined data types under a single name. We can access a complete structure as a whole or anyone of its members. The syntax of structure definition is as follows:

```
typedef struct
{ <predefined data type>   <member name>;
  <predefined data type>   <member name>;
  . . .                    . . . ;
} <structure type name>;
```

A **`<predefined data type>`** can be a standard data type in C, a previously defined structure type, or a pointer to one of those. **`<member name>`** is an identifier that we can use to access the individual members of a structure.

Definition of a new structure data type alone does not result in any memory space to be reserved for storage. Once it is defined, we can declare storage units or pointers for the new structure data type just as we declare variables or pointers for a standard data type. The following example defines a structured data type called "**`WaveformStruct`**".

```
// Define a new structure data type:
typedef struct
{ unsigned char        WaveType;   // waveform type (1 byte)
  unsigned short int   Nsample;    // number of samples (2 bytes)
  signed short int     Amplitude;  // amplitude in DAC units (2 bytes)
  unsigned long int    Period;     // period in ms (4 bytes)
} WaveformStruct;
```

The C compiler is ready to accept data storage or pointer declarations for the new data type, **`WaveformStruct`**, after this definition.

```
// Declare storage and pointer for the new structure type:
WaveformStruct     WFparam1;  // 9-byte storage for waveform parameters
WaveformStruct     WFparam2;  // another 9-byte storage
WaveformStruct    *WFparPtr;  // pointer to data type WaveformStruct
signed short int   OutGain;   // just another short integer
```

The compiler reserves **1+2+2+4=9** bytes of memory for **`WFparam1`**. The basic rules mentioned in the previous sections regarding the pointers for simple data types apply to the pointers declared with complex data structures. All pointers contain single address information regardless of the addressed data type, whether it is a 1-byte character or a 9-byte data structure. A pointer to a data structure keeps the address of the first byte of the first member of that data structure. Similarly, incrementing a pointer to a data structure advances the memory address stored in the pointer by the size of that data structure. As an example, the statement

```
WFparPtr ++;  // increment pointer to WaveformStruct
```

advances the memory address stored in **`WFparPtr`** by **9** since the **`WaveformStruct`** data type uses **9** bytes of memory.

C language supports statements to access an entire data structure or individual members of it. The following assignments copy 9 bytes of data between memory locations addressed directly or through pointers:

```
WFparam2 = WFparam1;   // copy all 9 bytes of the data structure
*WFparPtr = WFparam1;  // copy 9 bytes from WFparam1 to a target location
```

```
                       // starting at the address given in WFparPtr
WFparam2 = *WFparPtr;  // copy 9 bytes into WFparam2 from a location
                       // starting at the address given in WFparPtr
```

We can access the members of a structure directly or through a pointer:

```
WFparam1.WaveType = 3;          // write directly to member "WaveType"
OutGain = WFparam1.Amplitude;   // read directly from member "Amplitude"
WFparPtr = &WFparam1;           // get address of WFparam1
WFparPtr->WaveType = 3;         // write to member "WaveType"
                                // through a pointer to WaveformStruct
OutGain = WFparPtr->Amplitude;  // read from member "Amplitude"
                                // through a pointer to WaveformStruct
```

The <**member name**> specified in the structure data type definition identifies the member of the structure to be accessed. Different notations are used when structure members are accessed directly (a single decimal point precedes member name) and through a pointer ("**->**" are used instead of a decimal point).

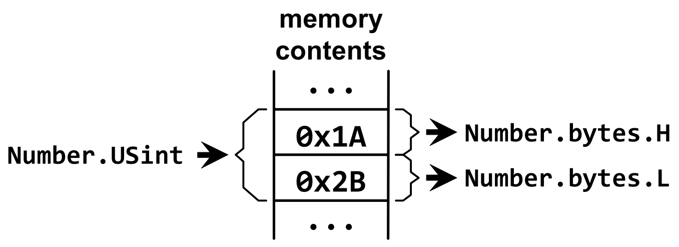
5.3.1 Unions

Unions allow different access ways to a memory location. Syntax rules for declaration and usage of unions are similar to those of structures, but the resultant memory organization is not the same. Members of a structure have their individual memory locations. Members of a union on the other hand, share the same memory location.

The following example defines a structure of two bytes, and then a union of an unsigned integer with the defined two-byte structure. The first member of the union, **USint**, is accessible as a single unsigned number. The same two-byte memory location is accessible as two detached bytes through the two-byte structure.

```
typedef struct     // type definition of a structure of two bytes
{ unsigned char  L;
  unsigned char  H;
} TwoBytes;

typedef union     // type definition of a short int <=> TwoBytes union
{ unsigned short int  USint;
  TwoBytes            bytes;
} USint_2byte;

USint_2byte    Number;  // declare a union of short int <=> TwoBytes

Number.USint = 0x1A2B;   // write a short integer to the union
PORTA = Number.bytes.H;  // send out the high byte = 0x1A
PORTA = Number.bytes.L;  // send out the low byte = 0x2B
```

The total memory reserved for the **Number** union is two bytes in this example. The first statement after the declaration writes a two-byte integer to the union memory. Other statements send the same data to **PORTA** one byte at a time.



The same output can be obtained with the following code that uses right shift operations:

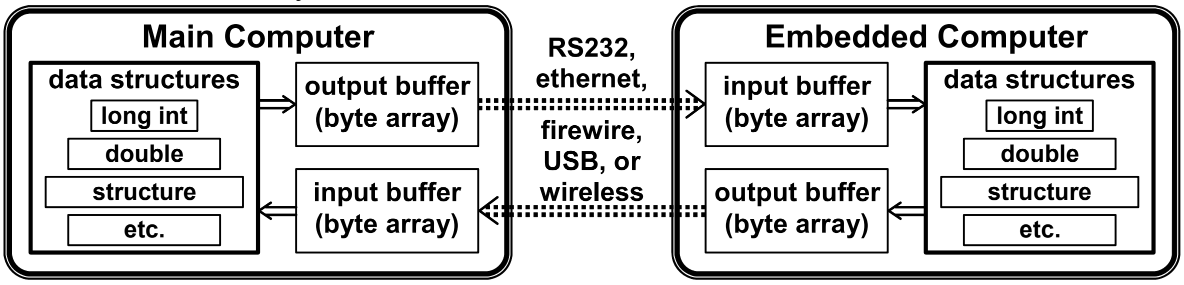
```
unsigned short int  Dout;  // declare a short int

Dout = 0x1A2B;
PortA = Dout >> 8;  // send out the high byte = 0x1A
                    // this will require many instructions
PortA = Dout;       // send out the low byte = 0x2B
```

Although this alternative source code looks simpler, the resultant executable code implements repeated right shift operations probably in a loop that will take a much longer time to complete. The first code example using the union is longer because of the type definitions. The type definitions are handled by the compiler, and the resultant executable code is more efficient. The unions in C language provide flexible and efficient usage of memory. The same efficiency can be obtained in assembly language but with some difficulty in programming.

5.4 Type Casting

Programs operate on many kind of data types, character strings, short integers, long integers, or structures that combine several of these. A microcontroller usually starts with a single sequence of bytes when it receives data from an external source, and it needs to convert the produced results into another sequence of bytes before sending them out. These are the typical examples of this process on an embedded system:

- Data are received or sent from/to another computer system through UART, USB, Ethernet or similar interfaces. The received data is stored in buffers as a sequence of bytes. Similarly, the data to be sent is stored in buffers or transmitted one byte at a time.


Main Computer
data structures
long int
double
structure
etc.
output buffer (byte array)
input buffer (byte array)
RS232, ethernet, firewire, USB, or wireless
Embedded Computer
input buffer (byte array)
output buffer (byte array)
data structures
long int
double
structure
etc.

- Input data from an ADC, a timer or another peripheral unit are read one byte at a time. If there is any buffering requirement, then the input data is stored as a sequence of bytes.
- Data sent to a DAC, a PWM controller or a similar output device are written one byte at a time, and/or buffered as a sequence of bytes.
- Different data types or data structures are stored in a single memory block that is allocated as a sequence of bytes.

Type casting allows changes in data types in executable statements. The word "***casting***" may sound more meaningful if we consider pouring a liquid metal into a mold to give it the desired shape. The syntax rule for type casting is simple: <u>the new data type enclosed in parenthesis precedes the variable name</u>. Consider the following declarations for the examples given below.

```
signed char          Byte1, Byte2;  // 1-byte integers
signed short int     Sinteger;      // 2-byte integer
signed long int      Linteger;      // 4-byte integer
unsigned char       *BufferPtr;     // pointer to single byte
signed long int     *LongPtr;       // pointer to long int
```

Following statement promotes 1-byte integers to 2-byte integers before the addition operation:

```
Sinteger = (signed short int)Byte1 + (signed short int)Byte2;
```

The promotion before the addition prevents an overflow as a result of the addition. The same result can be obtained by promoting one of the operands only:

```
Sinteger = (signed short int)Byte1 + Byte2;
```

In this last example, the C compiler will automatically promote `Byte2` to `signed short int` before performing the addition operation in the executable code.

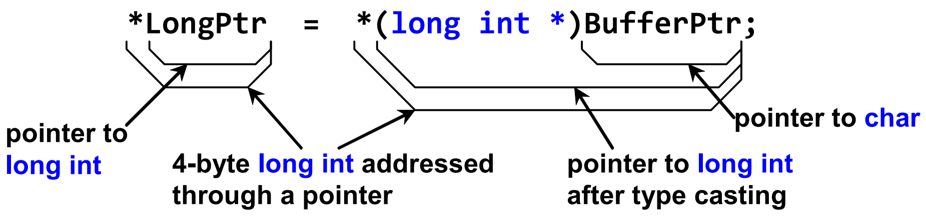
Type casting can also be applied to the memory access through pointers which is the main focus of this section. The following statements copy four bytes starting at the address given by **`BufferPtr`** into the variable **`Linteger`** or into the memory location pointed to by **`LongPtr`**:

```
Linteger = *(long int *)BufferPtr;
```

or

```
*LongPtr = *(long int *)BufferPtr;
```

The figure given below summarizes the logic behind the type casting operation. Basically, we tell the C compiler to treat **`BufferPtr`** as a pointer to a group of four bytes when we cast it into a new type with "**`(long int *)`**".


*LongPtr = *(long int *)BufferPtr;
pointer to long int
4-byte long int addressed through a pointer
pointer to long int after type casting
pointer to char

The statements given above can be used for transferring data from an input byte stream received from another computer system. The reverse operation is possible applying the sam<me type casting operation on the left hand side of the assignment. The following statements can be used for copying four bytes into an output buffer that will be sent out:

```
*(long int *)BufferPtr = Linteger;
```

or

```
*(long int *)BufferPtr = *LongPtr;
```

The same type casting operations can be used for the structure data type defined in the previous section:

```
// Copy 9 bytes from the buffer starting at BufferPtr:
WFparam1 = *(WaveformStruct *)BufferPtr;   // directly into data storage
*WFparPtr = *(WaveformStruct *)BufferPtr;  // through a pointer to data

// Copy 9 bytes into the buffer starting at BufferPtr:
*(WaveformStruct *)BufferPtr = WFparam1;   // directly from data storage
*(WaveformStruct *)BufferPtr = *WFparPtr;  // through a pointer to data
```

5.5 Notes to the Programmer

The pointers in C language and the type cast operations may seem confusing at the beginning, although all operations follow a few simple rules. Type casting provide the necessary flexibility and efficiency in accessing data while we can clearly define what we want to do. The basic rule of thumb is to interpret every asterisk correctly when we follow the changes from a data type to another data type, or from a pointer to another pointer. Now is a good time to mention some logical rules in type cast operations.

- **Numeric data types cast into numeric data types and pointers are cast into pointers.** If the target on the left hand side of an assignment is a pointer, then the right hand side should also produce a pointer type. Occasionally, pointers can be cast into unsigned numbers for data transmission or debugging purposes.
- **Type casting on left hand side of an assignment is applicable to pointers only**. A microprocessor cannot expand a `short int` into four bytes, or it cannot squeeze the four bytes of a `long int` into a single byte.
- **Number of bytes must be the same on both sides of an assignment** that access memory through pointers. An incomplete statement like,

  ```
  *BufferPtr = *LongPtr;
  ```

  leaves a question behind for the compiler as well as for anybody who reads this statement: *Are we trying to transfer one byte or four bytes?*

Pointers and the related operations described in this document are powerful features of the C programming language. They all come with serious responsibility just as any power that needs to be controlled. Programmers should be careful about the memory requirements especially when they are working on embedded systems where memory and other resources are minimized. The following statement copies 9 bytes into the buffer starting at the memory location pointed by `BufferPtr` as we have seen previously:

```
*(WaveformStruct *)BufferPtr = WFparam1;
```

It is the programmer's responsibility to make sure that there is enough space left in the buffer every time this statement is executed. If the `BufferPtr` points to the last 3 bytes of the memory area originally reserved for the buffer, then the remaining 6 bytes will be written outside the reserved area. These 6 bytes originating from `WFparam1` will overwrite the other variables stored in the data memory space with irrelevant values. The C compiler cannot detect this kind of programming errors that will cause unpredictable program behavior during execution because of the corrupt memory contents.

5.6 Compatibility Matters

The programs written in a standard programming language process all data types in the same way regardless of the computer system being used. A program written in standard C will produce the same results whether it is compiled for a microcontroller or for a personal computer. However, representations of the data types in the memory may vary from one computer system to another. The variations in memory representations does not cause a problem as long as the produced data remains in a single computer system. Compatibility problems come up when computer systems exchange data among themselves. The three common source of compatibility problems are the differences in compiler **default settings**, **byte order**, and **data structure alignment**.

Default Settings

At the beginning of this lecture, it is mentioned that the formal declarations, without omitting optional keywords, should be used to choose the optimum data type. Avoiding compatibility problems is another reason why we should use formal declarations. If the "`int`" keyword is used without the "`short`" or "`long`" specification to generate code for a 32-bit processor, then the compiler is likely to understand it as a 32-bit integer by default. If the "`int`" keyword is used alone on an 8-bit processor then it will probably give a 16-bit integer. Therefore, it is a good practice to use formal declarations not only for embedded code, but also in programs running on other computers that communicate with embedded systems.

Little or Big Endian

There are two alternatives to choose from for storing multiple byte numbers in memory. The first byte stored at the addressed memory location is the least significant 8 bits of a 16-bit (short int) or 32-bit (long int) number when the **little endian** byte order is used. Conversely, the **big endian** byte order places the most significant 8 bits at the first addressed byte.

16-bit Number: 0xA1B2

Little Endian:

address	data
N	B2
N+1	A1

Big Endian:

address	data
N	A1
N+1	B2

32-bit Number: 0x1A2B3C4D

Little Endian:

address	data
N	4D
N+1	3C
N+2	2B
N+3	1A

Big Endian:

address	data
N	1A
N+1	2B
N+2	3C
N+3	4D

The big endian order sounds logical, since we write numbers on the paper starting with the most significant digit. On the other hand, we should remember that we start with the least significant digit when we add numbers, just like an 8-bit processor that starts with the least significant byte while adding 16 or 32-bit integers.

The two byte order schemes are common in systems of all sizes. If two computer systems use different byte orders, then the low and high order bytes of all multiple byte numbers must be swapped when they exchange data. Note that an 8-bit processor hardware may support both little endian and big endian byte orders. A programmer should check for available assembler or compiler directives to change the byte order before attempting to solve this incompatibility problem by writing byte swap procedures.

Data Structure Alignment

The capacity of the data bus running between the processor and memory has been doubled several times since the introduction of early processors to answer the demand of increasing processing speed. The recent personal computers use 128-bit DDR SDRAM cards that transfer 16 times more data per cycle compared to 8-bit data bus of the first personal computer. Every single byte of memory has a distinct address regardless of the memory word size.

It is necessary to align variables with the word boundaries in order to fully utilize the large word size of a memory. The following figure gives examples of poor and efficient alignment for the 16-bit memory word size. Some memory locations can be left unused (shaded bytes in the figure) to minimize the number of memory access cycles. The basic idea is to confine variables in the minimum number of memory words.

Examples of poor alignment:

It takes 2 cycles to R/W 16-bit number.

address	data	
N	??	??
N+2	??	**1A**
N+4	**2B**	??
N+6	??	??

It takes 3 cycles to R/W 32-bit number.

address	data	
N	??	**1A**
N+2	**2B**	**3C**
N+4	**4D**	??
N+6	??	??

Alignment for efficient access:

It takes 1 cycle to R/W 16-bit number.

address	data	
N	??	??
N+2	??	▨
N+4	**1A**	**2B**
N+6	??	??

It takes 2 cycles to R/W 32-bit number.

address	data	
N	??	▨
N+2	**1A**	**2B**
N+4	**3C**	**4D**
N+6	??	??

Similar examples can be given for 32-bit memory word size: Four-byte `long int` variables should be aligned with the memory addresses that are integer multiples of 4. Similarly, 2-byte integers should have addresses that are integer multiples of 2. Compilers follow a set of rules when reserving storage for data structures to optimize memory access according to the specifications of computer systems. Refer to the article on "*data structure alignment*" on http://en.wikipedia.org for more information.

The insertion of unused bytes for optimized memory alignment is called **data structure padding**. Consider the following structure definition for example:

```
typedef struct
{ unsigned char     UC;
  signed short int  SSI;
  signed char       SC;
  signed long int   SLI;
} BadStruct;
```

If the alignment requirements are ignored, this structure requires **1+2+1+4=8** bytes of memory. On a typical 32-bit system, the C compiler will replace this structure definition with the following structure after the data structure padding required for alignment:

```
typedef struct
{ unsigned char     UC;
  char              DummyA;     // padding to align SSI
  signed short int  SSI;
  signed char       SC;
  char              DummyB[3];  // padding to align SLI
  signed long int   SLI;
} BadStruct;
```

After padding, the data structure occupies 12 bytes of memory, and the relative location of structure members are shifted as shown in the following figure. If the same data structure is defined on an 8-bit system, there will not be any alignment requirement. As a result, the actual memory representation of the same data structure will be different on the two systems. The data transfers that are arranged according to this data structure will cause errors unless the padding requirements are taken into account when receiving or transmitting data.

BadStruct without padding:

address	data			
N	UC	SSI-1	SSI-2	SC
N+4	SLI-1	SLI-2	SLI-3	SLI-4
N+8	??	??	??	??
N+12	??	??	??	??

BadStruct after padding:

address	data			
N	UC	DummyA	SSI-1	SSI-2
N+4	SC	DummyB	DummyB	DummyB
N+8	SLI-1	SLI-2	SLI-3	SLI-4
N+12	??	??	??	??

GoodStruct that requires no padding:

address	data			
N	UC	SC	SSI-1	SSI-2
N+4	SLI-1	SLI-2	SLI-3	SLI-4
N+8	??	??	??	??
N+12	??	??	??	??

The compiler directives that change the alignment requirements on the Windows systems are likely to cause errors in system calls, and they cannot be used most of the time. In some cases, simple rearrangement of structure members can eliminate the padding requirements. The data structure defined below has the same members of the structure given above, and all members are aligned properly without any padding.

```
typedef struct
{ unsigned char     UC;
  signed char       SC;
  signed short int  SSI;
  signed long int   SLI;
} GoodStruct;
```

Solution of compatibility problems brings up additional processing load during data transfers between computer systems. Generally, it is a better practice to implement the solutions on the system that has more resources and less timing restrictions.