INTRODUCTION

LECTURE I
TDDI11 Embedded Software

DEPT. COMPUTER AND INFORMATION SCIENCE (IDA)
LINKÖPINGS UNIVERSITET

COURSE INFORMATION

- Examiner: Ahmed Rezine
- Assistants: Xiaopeng Teng, Anton Hagel
- Administrator: Hanan Mohsen
- Lectures (8) and Computer lab sessions (13)
- Credits: 6 ECTS (Exam 2 ECTS, Labs 4 ECTS)

UPDATES

- See TDDI11 > Feedback & Updates (liu.se)

LABS

0. Warm up
1. Bit manipulation in C
2. Mixing C and assembly
3. I/O: polling versus interrupt driven
4. Preemptive threading
5. State machines

LABS

- Work in pairs. Register before April 7th on Webreg
  - (see "labs" under course homepage)
- Each pair solves the labs together.
- Solutions should be <u>individual</u> to each pair.
- Completed lab demonstration during scheduled lab sessions
- Both students in the pair are "present" and can answer all questions about any part of the solution

LECTURES

1. Introduction to embedded systems. Why C language?
2. Bit manipulation, mixing C and assembly
3. Pointers, structures and endianness
4. I/O programming
5. Concurrency and communication
6. State machines
7. Embedded design
8. Exam preparation

MATERIAL

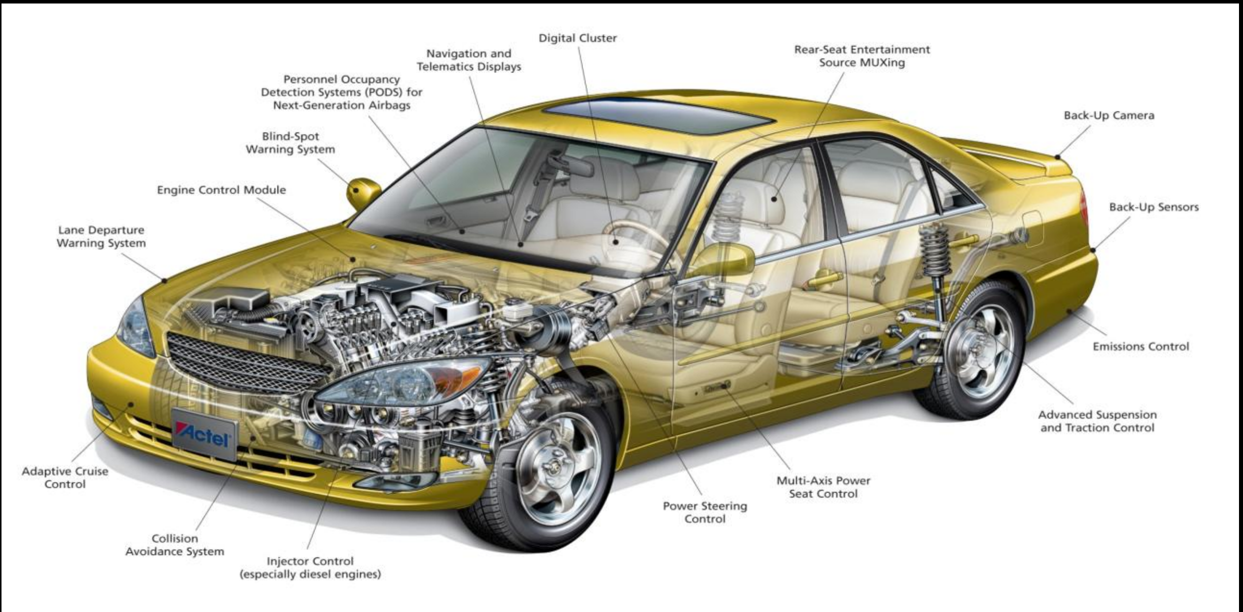
- Course homepage: Main source of information. E.g., pdf lectures, labs instructions
- Additional literature on subject: https://www.ida.liu.se/~TDDI11/info/literature.en.shtml

COMPUTING SYSTEMS

- Computing systems are everywhere
- Most of us think of “desktop” computers
  - PC’s, laptops, servers
- But there’s another type of computing system
  - Far more common...

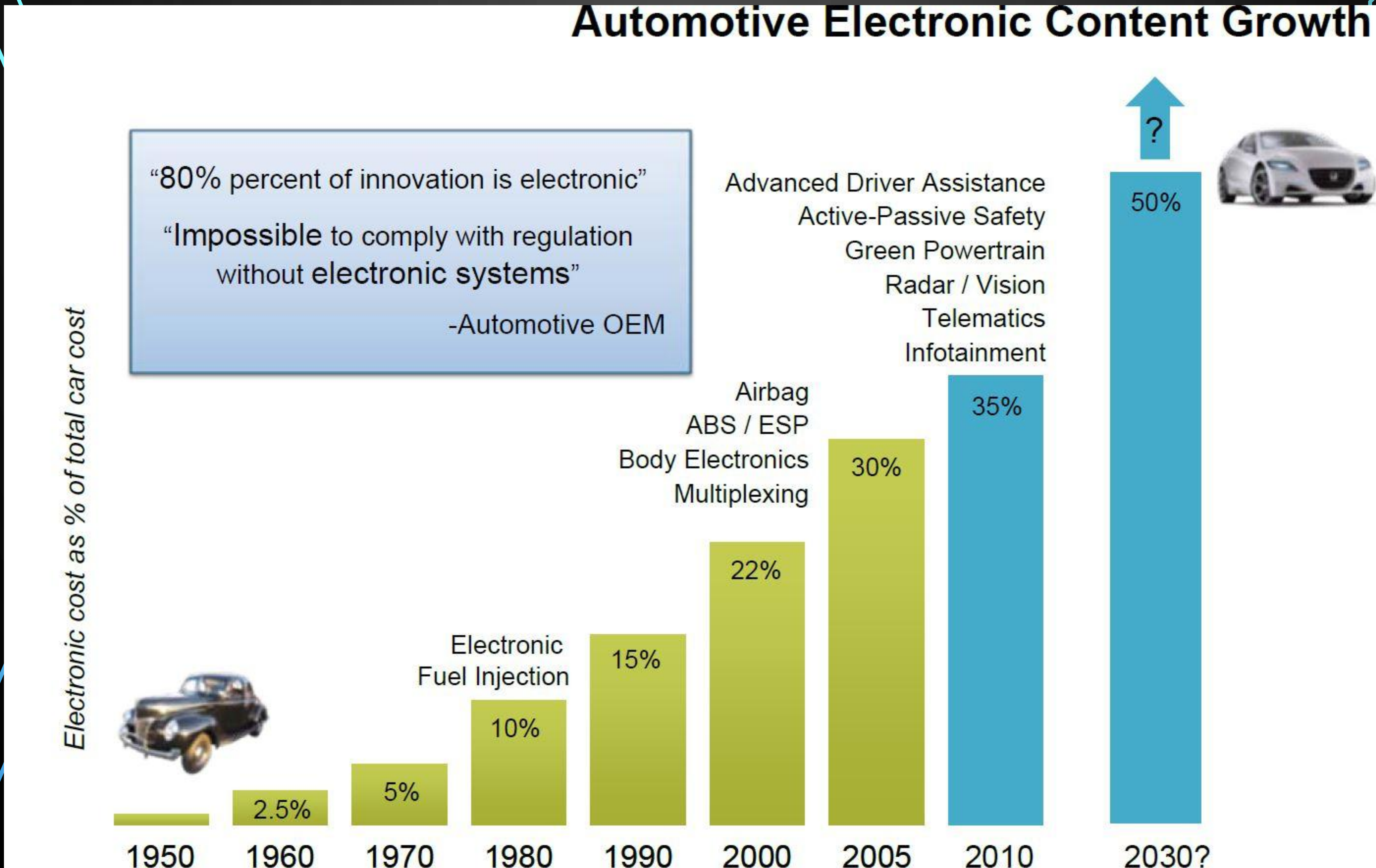
EMBEDDED SYSTEMS

- Computing systems embedded within electronic devices
- Billions of units produced yearly, versus millions of desktop units
- 50 or more per automobile with up to 100 million lines of code
- Nearly any computing system other than a desktop computer




Konnektor
Elektrode
Generator
ECG
76
/min
Pleth
75/min
99
105/69
(82)
35.9
17
Zero ALL
Auto On/Off
Start Cancel
Record Wave
Stop
NORMAL
HEAVY DUTY
BEDDING
RINSE+SPIN
QUICK WASH
PERM PRESS
htc
SAMSUNG

SIGNIFICANCE



EMBEDDED SYSTEMS EVOLUTION

- Present
  - 79% of all the processors are used in embedded systems
  - E.g., high-end cars contain more than 100 processors

- Future: Post-PC era
  - ❑ Cyber-physical systems
  - ❑ Internet of things
  - ❑ Wearables and implants to talk to "cloud"
  - ❑ Brain machine interfaces and body area networks

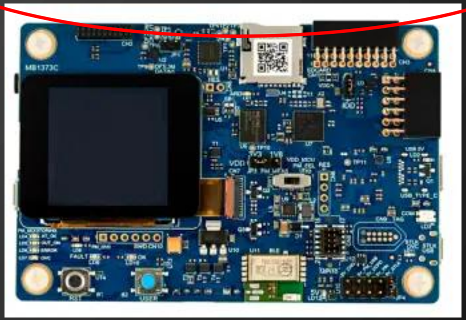
WHAT IS AN EMBEDDED SYSTEM?

- An **embedded system** is a:
  - Special-purpose computer system, part of a larger system which it controls.
  - Computing unit that interacts with the physical environment, via inputs and outputs

COMPONENTS OF AN EMBEDDED SYSTEM

- **Input** (Sensors)
  - Switches and buttons
  - Light, humidity, temperature
  - Microphone, camera

- **Microcontroller**



- **Output** (Actuators)
  - LED
  - Motor controller
  - Display
  - Relay

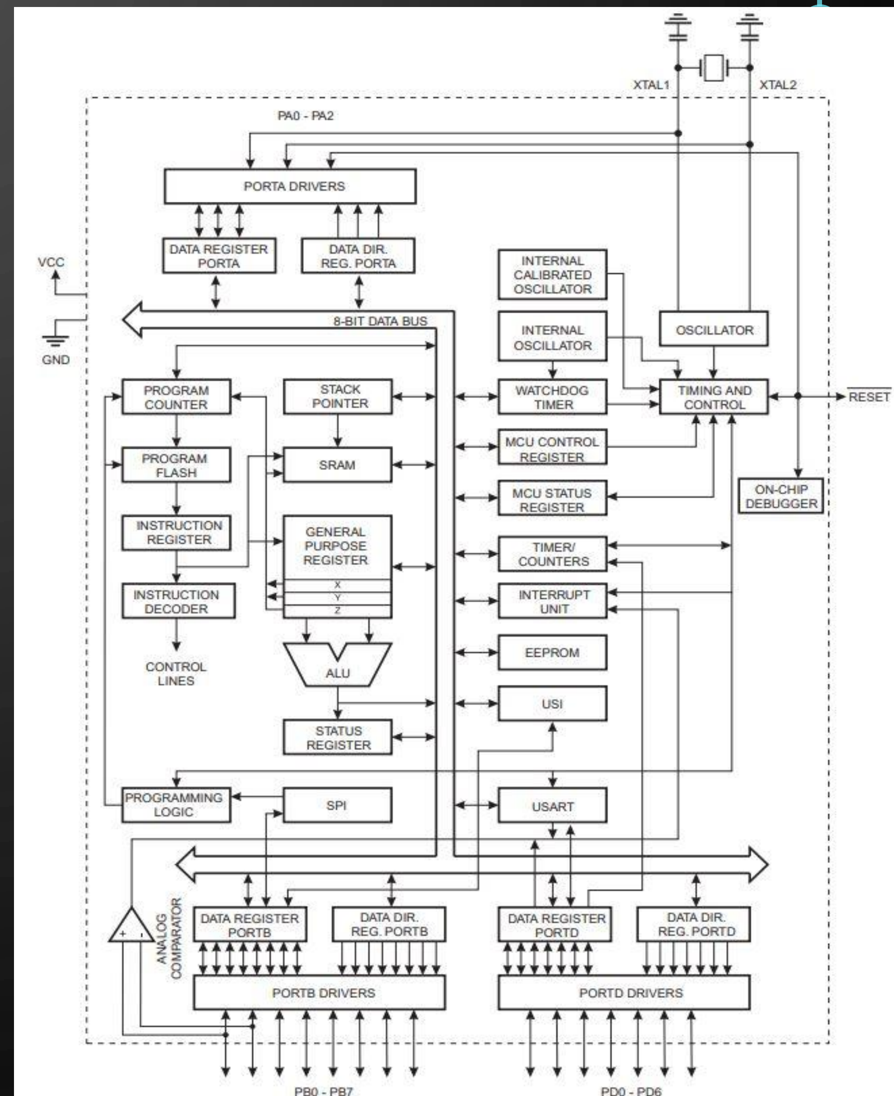
Sensors convert physical phenomena into digital input.

Actuators convert outputs to physical phenomena.

MICROCONTROLLER

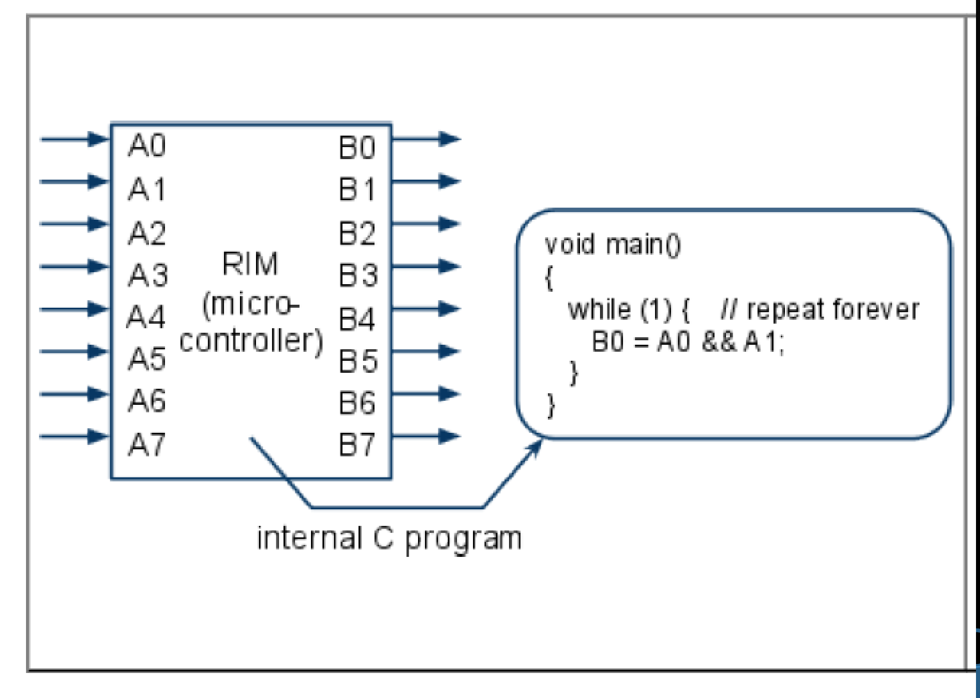
A programmable component that reads digital inputs and writes digital outputs according to some internally-stored program

Figure: A "PIC" microcontroller

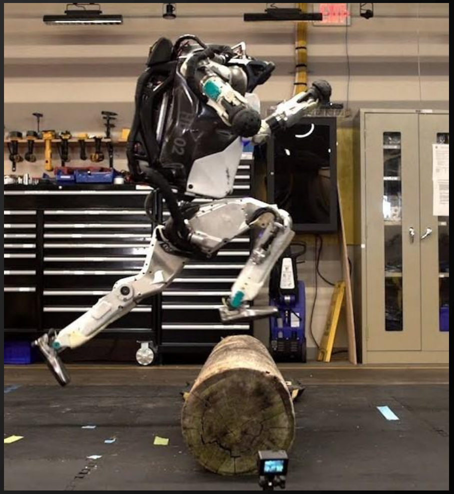


MICROCONTROLLER

- Capable of running software (e.g., C program)
- In this abstraction: 8 inputs and outputs – used by C program as implicit global variables
- This example shows an infinite while loop (repeat statements infinitely)



COMPLEXITY



- Physical reality is unpredictable
- Multiple functionalities often result in concurrency
- Current trends:
  - Connecting devices together (Internet of Things)
  - Adaptive, autonomous and learning systems

CRITICALITY

- Many of the application areas are safety-critical
  - Automotive, Avionics, Medicine, …
- Interaction with physical reality means
  - Reactivity (fast response time)
  - Real-time (guaranteed response time)
- Reliability
  - We expect devices to "just work"
  - Cannot fix software after shipping

FUNCTIONAL VS. NON-FUNCTIONAL REQUIREMENTS

- Functional requirements:
  - output as a function of input.
- Non-functional requirements:
  - time required to compute output;
  - size, weight, etc.;
  - power consumption;
  - reliability;
  - etc.

EMBEDDED VS. REAL-TIME SYSTEMS

- **Real-time system:**

The correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced

  - **Hard real-time**: missing deadline causes failure
  - **Soft real-time**: missing deadline results in degraded performance

- A real-time system is not necessarily embedded
- An embedded system is not necessarily real-time

SUMMARY

- Embedded system definition
- Special-purpose. All around us: transportation, medical equipment, home appliances, ...
- Interacts with physical environment through inputs and outputs
- Challenges:
  - Complexity: multiple algorithms, concurrency
  - Scarcity of resources: cost, power, size, weight,...
  - Criticality: safety-critical, real-time, reliable

LECTURES

1. Introduction to embedded systems. Why C language?
2. Bit manipulation, mixing C and assembly
3. Pointers, structures and endianness
4. I/O programming
5. Concurrency and communication
6. State machines
7. Embedded design
8. Exam preparation

C FOR EMBEDDED SYSTEMS

The Course covers C constructs that are frequently used in embedded software development:

- **Preprocessor directives**: Informs the compiler about the hardware
- **Mixing C and assembly**: Often a necessity in embedded systems
- **Pointers**: Used to access memory and input and output devices
- **Bit manipulation**: Used to handle hardware-level details, input and output
- **Structures, unions**: In the context of pointers and bit manipulation

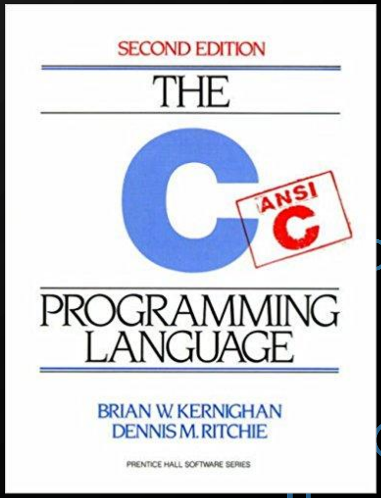
C or C++ knowledge is a prerequisite for this course

HISTORY OF C

- Designed and developed by Dennis Ritchie in beginning of 70s at Bell Labs.
- Used to develop UNIX (with Ken Thompson)
- Used to write modern operating systems
- Hardware independent (portable)

C STANDARDIZATION

- Many slight variations of C existed, and were incompatible
- Committee formed to create an "unambiguous, machine-independent" definition
- Standard created in 1989 (ISO C), updated in 1999 (C99), in 2011 (C11) and in 2018 (C18).



WHY USE C FOR WRITING EMBEDDED SOFTWARE?

- Small and simple to learn
- Available for almost all currently used processors
- C is a very "low-level" high-level language
- It gives embedded programmers a high degree of hardware control without sacrificing the benefits of high-level languages

PREPROCESSING DIRECTIVES

- Preprocessing
  - Occurs before a program is compiled
  - Inclusion of other files
  - Definition of symbolic constants and macros
  - Conditional compilation of program code
  - Conditional execution of preprocessor directives
- Format of preprocessor directives
  - Lines begin with #

THE #INCLUDE PREPROCESSOR DIRECTIVE

- Copy of a specified file included in place of the directive
  - `#include <filename>`

    Searches standard library for file

    Use for standard library files
  - `#include "filename"`

    Searches current directory, then standard library

    Use for user-defined files

- Used for:
  - Programs with multiple source files to be compiled together
  - Header file – has common declarations and definitions (structures, function prototypes)

    #include statement in each file

THE #DEFINE PREPROCESSOR DIRECTIVE

Symbolic constants

- #define
  - Preprocessor directive used to create symbolic constants and macros
  - Symbolic constants
    - When program compiled, all occurrences of symbolic constant replaced with replacement text
  - Format

```
#define identifier replacement-text
```

  - Example:

```
#define PI 3.14159
```

  - Everything to right of identifier replaces text

```
#define PI = 3.14159
```

    - Replaces “PI” with "= 3.14159”

THE #DEFINE PREPROCESSOR DIRECTIVE

- Macro
  - Operation defined in #define
  - A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
  - Performs a text substitution – no data type checking
  - The macro

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

would cause

```
area = CIRCLE_AREA( 4 );
```

to become

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

EXAMPLES: PITFALLS

It is left as an exercise to find out what (may) become wrong with the definition below:

```
#define POW(x) x*x
#define CIRCLE_AREA( x )  PI *  x  * x
#define RECTANGLE_AREA( x, y )   x  *  y
```

reference:

http://gcc.gnu.org/onlinedocs/cpp/macros.html

CONDITIONAL COMPILATION

- Control preprocessor directives and compilation
- sizeof, enumeration constants cannot be evaluated in preprocessor directives
- Structure similar to if

```
#if ! defined (NULL)
   #define NULL 0
#endif
```

- Determines if symbolic constant NULL has been defined
  – If NULL is defined, defined (NULL) evaluates to 1
  – If NULL is not defined, this function defines NULL to be 0
- Every #if must end with #endif
    - #ifdef short for #if defined(name)
    - #ifndef short for #if !defined(name)

CONDITIONAL COMPILATION, CONT.

- Use for commenting out code
- C does not allow nested comments

```
/* First layer
    /* Second layer */
*/
```

- You can use the #if .. #endif combination to cause the preprocessor to avoid compiling any portion of your code by using a condition that will never be true.

```
#if 0
    code commented out
#endif
```

  - To enable code, change 0 to 1

CONDITIONAL COMPILATION, CONT.

- Other statements
  - #elif – equivalent of else if in an if statement
  - #else – equivalent of else in an if statement

CONDITIONAL COMPILATION, CONT.

- Debugging

```
#define DEBUG  1
#if DEBUG
     printf(…);
#endif
```

  - Defining DEBUG to 1 enables code
  - After code corrected, remove #define statement
  - Debugging statements are now ignored

THE #ERROR PREPROCESSOR DIRECTIVES

- #error tokens
  - Tokens are sequences of characters separated by spaces
    - "I like C" has 3 tokens
  - Displays a message with specified tokens as an error message
  - Stops preprocessing and prevents program compilation
- The directive ‘#error’ causes the preprocessor to report a fatal error. The tokens forming the rest of the line following ‘#error’ are used as the error message.
  - E.g.,

```
#if !defined(FOO) && defined(BAR)
    #error "BAR requires FOO."
#endif
```

THE # AND ## OPERATORS

- #
  - Causes a replacement text token to be converted to a string surrounded by quotes
  - The statement

```
#define HELLO( x ) printf( “Hello, ” #x  “\n” );
```

would cause

```
HELLO( John )
```

to become

```
printf( “Hello, ” “John” “\n” );
```

THE # AND ## OPERATORS, CONT.

- ##
  - Concatenates two tokens
  - The statement

```
#define TOKENCONCAT( x, y )  x ## y
```

would cause

```
TOKENCONCAT( O, K )
```

to become

```
OK
```

PREDEFINED SYMBOLIC CONSTANTS

- Useful predefined symbolic constants
  - Cannot be used in #define or #undef

Symbolic constant	Description
__LINE__	The line number of the current source code line (constant integer)
__FILE__	The path or name of the file (e.g., "/usr/local/include/myheader.h")
__DATE__	The date the source file is compiled (e.g., "Jan 19 2001")
__TIME__	Time the source file is compiled (e.g., "08:22:17").

EXAMPLES: MACRO DEFINITION AND USE

```
#define SIZE 128

#define POW(x) ((x)*(x))

#define DEBUG(format, ...) printf(format, ## __VA_ARGS__)

#define DUMP(int_var) printf(”%s = %d\n”, #int_var, int_var)

#define WHERE printf(”’%s’ at %d\n”, __FILE__, __LINE__)


int main(){

  int array[SIZE];

  POW(array[0] + array[1]);

  /* Disable all DEBUG by changing macro */

  DEBUG(”%s\n”, ”I reached the top”);

  /* Easy to get nice print of variable. */

  DUMP(array[4]);

  /* Prints file and line. */

  WHERE;

}
```