HIGH-LEVEL LANGUAGE INTERFACES

CS 16: Assembly Language Programming for the IBM PC and Compatibles
Objectives

- Introduce high-level language interfaces
- Look at inline assembly code
- Link 32-Bit assembly language code to C/C++
Why link Assembly Language and High-Level Language (HLL) Programs?

- Use high-level language for overall project development
  - Relieves programmer from low-level details
- Use assembly language code
  - Speed up critical sections of code
  - Access nonstandard hardware devices
  - Write platform-specific code
  - Extend the HLL's capabilities
Considerations when calling assembly language procedures from high-level languages:

- Both must use the same naming convention (rules regarding the naming of variables and procedures)
- Both must use the same memory model, with compatible segment names
- Both must use the same calling convention
CALLING CONVENTION

- Identifies specific registers that must be preserved by procedures
- Determines how arguments are passed to procedures: in registers, on the stack, in shared memory, etc.
- Determines the order in which arguments are passed by calling programs to procedures
**CALLING CONVENTION (2)**

- Determines whether arguments are passed by value or by reference
- Determines how the stack pointer is restored after a procedure call
- Determines how functions return values
An external identifier is a name that has been placed in a module’s object file in such a way that the linker can make the name available to other program modules.

The linker resolves references to external identifiers, but can only do so if the same naming convention is used in all program modules.
Assembly language source code that is inserted directly into a HLL program

Compilers such as Microsoft Visual C++ and Borland C++ have compiler-specific directives that identify inline ASM code

Efficient inline code executes quickly because CALL and RET instructions are not required

Simple to code because there are no external names, memory models, or naming conventions involved

Decidedly not portable because it is written for a single platform
Can be placed at the beginning of a single statement
Or, it can mark the beginning of a block of assembly language statements
Syntax:

```
__asm statement
__asm {
    statement-1
    statement-2
    ...
    statement-n
}
```
**COMMENTING STYLES**

- All of the following comment styles are acceptable, but the latter two are preferred:
  
  ```asm
  mov esi,buf ; initialize index register
  mov esi,buf // initialize index register
  mov esi,buf /* initialize index register */
  ```
You can do the following...

- Use any instruction from the Intel instruction set
- Use register names as operands
- Reference function parameters by name
- Reference code labels and variables that were declared outside the asm block
You can do the following... (2)

- Use numeric literals that incorporate either assembler-style or C-style radix notation
- Use the PTR operator in statements such as inc BYTE PTR [esi]
- Use the EVEN and ALIGN directives
- Use LENGTH, TYPE, and SIZE directive
YOU CANNOT DO THE FOLLOWING...

- Use data definition directives such as DB, DW, or BYTE
- Use assembler operators other than PTR
- Use STRUCT, RECORD, WIDTH, and MASK
YOU CANNOT DO THE FOLLOWING... (2)

- Use the OFFSET operator (but LEA is ok)
- Use macro directives such as MACRO, REPT, IRC, IRP
- Reference segments by name.
  - You can, however, use segment register names as operands
REGISTER USAGE

- In general, you can modify EAX, EBX, ECX, and EDX in your inline code because the compiler does not expect these values to be preserved between statements.
- Conversely, always save and restore ESI, EDI, and EBP.
FILE ENCRYPTION EXAMPLE

- Reads a file, encrypts it, and writes the output to another file
- The TranslateBuffer function uses an __asm block to define statements that loop through a character array and XOR each character with a predefined value
// ENCODE2.CPP

#include <iostream.h>
#include <fstream.h>

// Translate a buffer of <count> bytes, using an encryption
// character <eChar>. Uses an XOR operation (ASM function).

const int BUFSIZE = 200;
char buffer[BUFSIZE];
```cpp
int main()
{
    unsigned count;    // character count
    unsigned short encryptCode;
    cout << "Encryption code [0-255]? ";
    cin >> encryptCode;
    unsigned char encryptChar = (unsigned char) encryptCode;

    ifstream infile( "infile.txt", ios::binary );
    ofstream outfile( "outfile.txt", ios::binary );

    cout << "Reading INFILE.TXT and creating OUTFILE.TXT...\n";
```
while (!infile.eof() )
{
    infile.read(buffer, BUFSIZE );
    count = infile.gcount();

    __asm {
        lea esi,buffer
        mov ecx,count
        mov al, encryptChar
    }
}
L1:
    xor [esi], al
    inc  esi
    Loop L1
} // asm
out_file.write(buffer, count);
}
return 0;
}
Basic Structure - Two Modules

- The first module, written in assembly language, contains the external procedure
- The second module contains the C/C++ code that starts and ends the program

The C++ module adds the extern qualifier to the external assembly language function prototype.

The "C" specifier must be included to prevent name decoration by the C++ compiler:

```c
extern "C" functionName( parameterList );
```
HLL compilers do this to uniquely identify overloaded functions.

A function such as:

```c
int ArraySum( int * p, int count )
```

would be exported as a decorated name that encodes the return type, function name, and parameter types.

For example:

```c
int_ArraySum_pInt_int
```

The problem with name decoration is that the C++ compilers vary in the way they decorate function names.
Use assembly language to optimize sections of applications written in high-level languages
- Inline asm code
- Linked procedures

Naming conventions, name decoration

Calling convention determined by HLL program

OK to call C functions from assembly language