
High-Level Language Interface

Chapter 13
S. Dandamudi

High-Level Language Interface

- Why program in mixed-mode?
 - * Focus on C and assembly
- Overview of compiling mixed-mode programs
- Calling assembly procedures from C
 - * Parameter passing
 - * Returning values
 - * Preserving registers
 - * Publics and externals
 - * Examples
- Calling C functions from assembly
- Simplified calling mechanisms
 - * The ARG directive
 - * Avoiding explicit specification of underscores
 - * Extended CALL instruction
- Inline assembly code

Why Program in Mixed-Mode?

- Pros and cons of assembly language programming
 - * Advantages:
 - » Access to hardware
 - » Time-efficiency
 - » Space-efficiency
 - * Problems:
 - » Low productivity
 - » High maintenance cost
 - » Lack of portability
- As a result, some programs are written in mixed-modern (e.g., system software)

1998

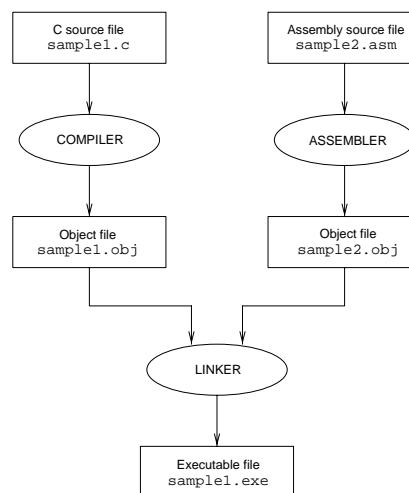
© S. Dandamudi

HLL Interface: Page 3

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Compiling Mixed-Mode Programs

- We use C and assembly mixed-mode programming
- Our emphasis is on the principles
- Can be generalized to any type of mixed-mode programming
- To compile
`bcc sample1.c sample.asm`



1998

© S. Dandamudi

HLL Interface: Page 4

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Calling Assembly Procedures from C

Parameter Passing

- Stack is used for parameter passing
- Two ways of pushing arguments onto the stack
 - * Left-to-right
 - » Most languages including Basic, Fortran, Pascal use this method
 - » These languages are called *left-pusher* languages
 - * Right-to-left
 - » C uses this method
 - » These languages are called *right-pusher* languages

1998

© S. Dandamudi

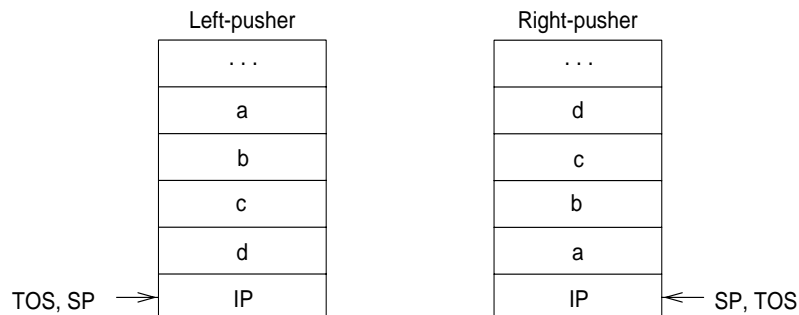
HLL Interface: Page 5

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Calling Assembly Procedures from C (cont'd)

Example:

sum(a,b,c,d)



1998

© S. Dandamudi

HLL Interface: Page 6

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Calling Assembly Procedures from C (cont'd)

Returning Values

- Registers are used to return values

Return value type	Register used
char, short, int (signed/unsigned)	AX
long (signed/unsigned)	DX:AX
near pointer far pointer	AX DX:AX

1998

© S. Dandamudi

HLL Interface: Page 7

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Calling Assembly Procedures from C (cont'd)

Preserving Registers

- The following registers must be preserved
BP, SP, CS, DS, SS
- In addition, if register variables are enabled,
SI and DI
should also be preserved.
- Since we never know whether register variables
are enabled or not, it is a good practice to preserve
BP, SP, CS, DS, SS, SI and DI

1998

© S. Dandamudi

HLL Interface: Page 8

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Calling Assembly Procedures from C (cont'd)

Publics and External

- Mixed-mode programming involves at least two program modules
 - » One C module and one assembly module
- We have to declare those functions and procedures that are not defined in the same module as external
 - » `extern` in c
 - » `extrn` in assembly
- Those procedures that are accessed by another modules as public

Calling Assembly Procedures from C (cont'd)

Underscores

- In C, all external labels start with an underscore
 - » C and C++ compilers automatically append the required underscore on all external functions and variables
- You must make sure that all assembly references to C functions and variables begin with underscores
- Also, you should begin all assembly functions and variables that are made public and referenced by C code with underscores

Calling C Functions from Assembly

- Stack is used to pass parameters (as in our previous discussion)
- Similar mechanism is used to pass parameters and to return values
- Since C makes the calling procedure responsible for clearing the stack of the parameters, make sure to clear the parameters after the **call** instruction as in

add SP, 4

on line 45 in the example program

Simplified Calling Mechanisms

The ARG Directive

- By using ARG directive, we can let the assembler calculate the offset values of the parameters on the stack
- Arguments in ARG directive are listed in the same order as in the C call
 - * All arguments should be listed in a single line
 - * If necessary, use '\ ' to extend the ARG line beyond 80 characters
 - * If type is not specified, TASM assumes WORD for 16-bit models, DWORD for 32-bit models

Simplified Calling Mechanisms (cont'd)

Avoiding explicit specification of underscores

- We can let the assembler prefix the required underscore on all external functions and variables
- We need to let the assembler know that we are using C language
 - » We do this by using
PUBLIC C
instead of **PUBLIC** (see line 10 in the example program)
- We can use a similar method for **EXTRN** as well (i.e., **EXTRN C**)
 - » see line 8 in the example program

1998

© S. Dandamudi

HLL Interface: Page 13

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Simplified Calling Mechanisms (cont'd)

Extended CALL Instruction

- This instruction relieves us from pushing the arguments onto the stack before a procedure call
 - » Assembler will insert the necessary push instructions
- The syntax is
CALL destination [language[,arg1]...]
language is C, CPP, Pascal, Fortran, etc.
- Extended CALL does three things:
 - » Pushes the arguments in the correct order (right or left pushing based on the language specified)
 - » Prefixes an underscore if required (as in C)
 - » Clears the stack of the arguments if needed (as in C)

1998

© S. Dandamudi

HLL Interface: Page 14

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Inline Assembly Code

- Assembly language statements are embedded into the C code
 - » Separate assembly module is not necessary
- Assembly statements are identified by placing the keyword **asm**

```
asm xor AX,AX; mov AL,DH
```

- We can use braces to compound several assembly statements

```
asm {  
    xor AX,AX  
    mov AL,DH  
}
```

1998

© S. Dandamudi

HLL Interface: Page 15

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Inline Assembly Code (cont'd)

Example

Get date interrupt service

- * Uses interrupt 21H service
- * Details:

Input:

AH = 2AH

Returns:

AL = day of the week (0=Sun, 1=Mon,...)

CX = year (1980 - 2099)

DH = month (1=Jan, 2=Feb, ...)

DL = day of the month (1-31)

1998

© S. Dandamudi

HLL Interface: Page 16

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

Inline Assembly Code (cont'd)

Compiling inline Assembly Programs

Two ways:

* TASM method

- » Convert C code into assembly language and then invoke TASM to produce .OBJ file
- » Can use **-B** compiler option to generate assembly file
- » Alternatively, can include

#pragma inline

at the beginning of the C file to instruct the compiler to use the **-B** option

* BASM method

- » Uses the built-in assembler (BASM) to assemble **asm** statements
- » Restricted to 16-bit instructions (i.e., cannot use 486 or Pentium instructions)

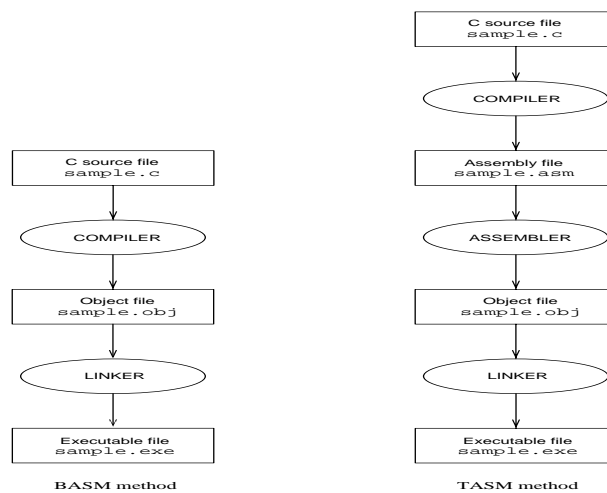
1998

© S. Dandamudi

HLL Interface: Page 17

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.

Inline Assembly Code (cont'd)



1998

© S. Dandamudi

HLL Interface: Page 18

To be used with S. Dandamudi, "Introduction to Assembly Language Programming." Springer-Verlag, 1998.