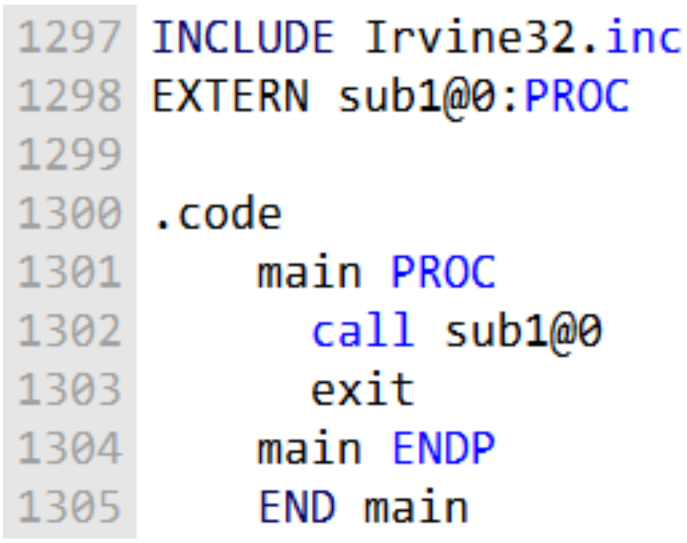
Calling External Procedures

To call an external procedure in MASM, you use the EXTERN directive.

The EXTERN directive tells the assembler that the procedure is defined in another module and gives the procedure's name and stack frame size.

The following example shows how to call an external procedure named sub1():



```
INCLUDE Irvine32.inc
EXTERN sub1@0:PROC

.code
    main PROC
      call sub1@0
      exit
    main ENDP
    END main
```

The **@0 suffix** at the end of the procedure name indicates that the procedure **does not have any parameters.**

If the procedure has parameters, you must include the **stack frame size** in the EXTERN directive.

The stack frame size is the total amount of stack space that the procedure uses to store its parameters and local variables.

The following example shows how to call an external procedure named AddTwo(), which has two doubleword parameters:

```
INCLUDE Irvine32.inc
EXTERN AddTwo@8:PROC

.code
    main PROC
      call AddTwo@8
      exit
    main ENDP
    END main
```

The @8 suffix at the end of the procedure name indicates that the procedure uses 8 bytes of stack space for its parameters.

You can also use the PROTO directive in place of the EXTERN directive. The PROTO directive tells the assembler about the prototype of the procedure, including its name and the number and types of its arguments.

The following example shows how to use the PROTO directive to declare the prototype of the AddTwo() procedure:

```
INCLUDE Irvine32.inc
PROTO AddTwo,
val1:DWORD,
val2:DWORD

.code
    main PROC
      call AddTwo
      exit
    main ENDP
    END main
```

The PROTO directive tells the assembler that the AddTwo() procedure has two doubleword parameters.

When the assembler sees a call to the AddTwo() procedure, it can check to make sure that the correct number of arguments are passed to the procedure.

The EXTERN and PROTO directives are both used to call external procedures.

The EXTERN directive is simpler to use, but the PROTO directive provides more information to the assembler, which can help to prevent errors.

Which directive should you use?

If you are only calling a few external procedures and you are not concerned about the performance of your program, then you can use the EXTERN directive.

If you are calling a large number of external procedures or if you are concerned about the performance of your program, then you should use the PROTO directive.

The PROTO directive provides more information to the assembler, which can help to optimize the code.

==

Using Variables and Symbols across Module Boundaries:

Exporting Variables and Symbols

==

What is EXTERNDEF?

EXTERNDEF is a directive that combines the functionality of the PUBLIC and EXTERN directives. It can be used to export variables and symbols from one module and import them into another module.

How to use EXTERNDEF?

To use EXTERNDEF, you first need to create an include file that contains the EXTERNDEF declarations for the variables and symbols that you want to share. For example, the following include file defines two variables, count and SYM1:

```
; vars.inc
EXTERNDEF count:DWORD, SYM1:ABS
```

Then, you can include the include file in any module that needs to access the variables or symbols. For example, the following module exports the count and SYM1 variables:

```
; sub1.asm
.386
.model flat,STDCALL
INCLUDE vars.inc
SYM1 = 10
.data
    count DWORD 0
    END
```

Finally, you can import the variables or symbols into any module that needs to use them. For example, the following module imports the count and SYM1 variables and uses them to calculate a value:

```
; main.asm
.386
.model flat,stdcall
.stack 4096
ExitProcess proto, dwExitCode:dword
INCLUDE vars.inc
.code
    main PROC
    mov
    count,2000h
    mov
    eax,SYM1
    INVOKE ExitProcess,0
    main ENDP
    END main
```

Benefits of using EXTERNDEF

There are several benefits to using EXTERNDEF to share variables and symbols across module boundaries:

It makes it easy to share variables and symbols between modules. It helps to reduce the amount of duplicate code.

It makes the code more modular and reusable. Conclusion
EXTERNDEF is a powerful directive that can be used to share variables and symbols across module boundaries.

It is a good practice to use EXTERNDEF to share variables and symbols between modules, as it makes the code more modular and reusable.

Let's make our program modular and see these concepts in action:

===========================

ArraySum program

===========================

The ArraySum program, first presented in Chapters before, is a good example of a multimodule program. The program can be divided into the following modules:

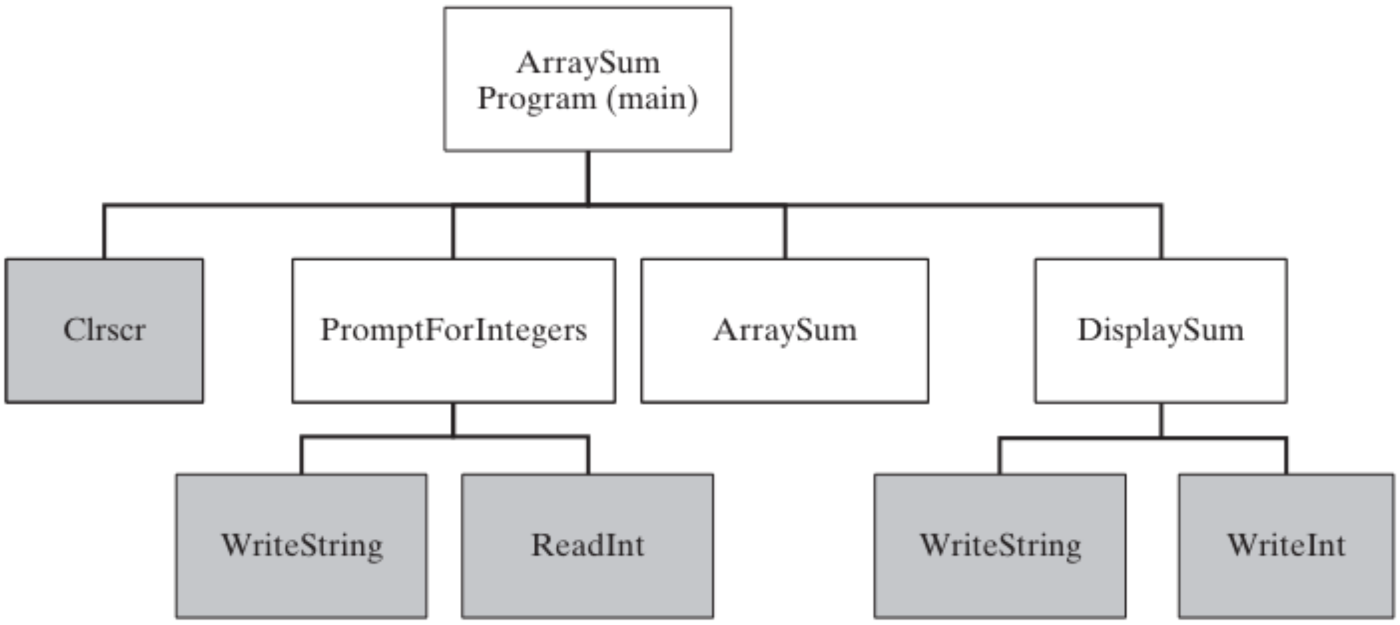
main.asm: The startup module, which calls the other modules to perform the program's tasks.

promptforintegers.asm: Prompts the user to enter an array of integers and reads the integers from the console.

arraysum.asm: Calculates the sum of the integers in the array.

writeinteger.asm: Writes an integer to the console.

The following diagram shows the structure chart of the ArraySum program:



=============================

Prompt for Integers

=============================

```
; Prompt For Integers (_prompt.asm)
INCLUDE Irvine32.inc
.code
    ;-----------------------------------------------------
    PromptForIntegers PROC
    ; Prompts the user for an array of integers and fills
    ; the array with the user's input.
    ; Receives:
    ; ptrPrompt: PTR BYTE - prompt string
    ; ptrArray: PTR DWORD - pointer to array
    ; arraySize: DWORD - size of the array
    ; Returns: nothing
    ;------------------------------------------------------
    arraySize EQU [ebp+16]
    ptrArray EQU [ebp+12]
    ptrPrompt EQU [ebp+8]
    enter 0,0
    pushad
    ; save all registers
    mov ecx,arraySize
    cmp ecx,0
    ; array size != 0?
    jle L2
    ; yes: quit
    mov edx,ptrPrompt
    ; address of the prompt
    mov esi,ptrArray
    L1:
    call WriteString
    ; display string
    call ReadInt
    ; read integer into EAX
    call Crlf
    ; go to next output line
    mov [esi],eax
    ; store in array
    add esi,4
    ; next integer
    loop L1
    L2:
    popad
    ; restore all registers
    leave
    ret
    12
    ; restore the stack
    PromptForIntegers ENDP
    end
```

The prompt.asm file contains the source code for the PromptForIntegers() procedure. This procedure prompts the user for an array of integers and fills the array with the user's input.

- The **PromptForIntegers()** procedure takes three parameters:
- **ptrPrompt:** A pointer to the prompt string.
- **ptrArray:** A pointer to the array.
- **arraySize:** The size of the array.

The PromptForIntegers() procedure works as follows:

It saves all of the registers.

It compares the array size to 0. If the array size is 0, the procedure exits. It displays the prompt string using the WriteString() procedure. It reads an integer from the console using the ReadInt() procedure. It stores the integer in the array.

It increments the array pointer. It repeats steps 3-6 until all of the integers have been read. It restores all of the registers. It leaves the procedure. Here is a more detailed explanation of each step:

Step 1: Save all of the registers
The PromptForIntegers() procedure saves all of the registers because it needs to use them and it does not want to overwrite any of the values that are in the registers when the procedure is called.

Step 2: Compare the array size to 0
The PromptForIntegers() procedure checks the array size to make sure that it is not 0. If the array size is 0, the procedure exits. This is because it does not make sense to prompt the user for an array of integers if the array is empty.

Step 3: Display the prompt string
The PromptForIntegers() procedure displays the prompt string using the WriteString() procedure. The WriteString() procedure is a library procedure that writes a string to the console.

Step 4: Read an integer from the console
The PromptForIntegers() procedure reads an integer from the console using the ReadInt() procedure. The ReadInt() procedure is a library procedure that reads an integer from the console and stores it in the EAX register.

Step 5: Store the integer in the array
The PromptForIntegers() procedure stores the integer in the array by

moving the EAX register to the array element that is pointed to by the array pointer.

Step 6: Increment the array pointer

The PromptForIntegers() procedure increments the array pointer so that it points to the next element in the array.

Step 7: Repeat steps 3-6 until all of the integers have been read

The PromptForIntegers() procedure repeats steps 3-6 until all of the integers have been read. This is done by using the loop instruction. The loop instruction repeats a block of instructions until a specified condition is met. In this case, the condition is that the array pointer is not equal to the end of the array.

Step 8: Restore all of the registers

The PromptForIntegers() procedure restores all of the registers that it saved in step 1.

Step 9: Leave the procedure

The PromptForIntegers() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The **PromptForIntegers() procedure** is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses library procedures to perform common tasks, such as writing a string to the console and reading an integer from the console.

=============================

ArraySum program

=============================

```
; ArraySum Procedure (_arraysum.asm)
INCLUDE Irvine32.inc
.code
    ;-------------------------------------------------------
    ArraySum PROC
    ;
    ; Calculates the sum of an array of 32-bit integers.
    ; Receives:
    ; ptrArray - pointer to array
    ; arraySize - size of array (DWORD)
    ; Returns: EAX = sum
    ;-----------------------------------------------------
    ptrArray EQU [ebp+8]
    arraySize EQU [ebp+12]
    enter 0,0
    push ecx
    ; don't push EAX
    push esi
    mov eax,0
    ; set the sum to zero
    mov esi,ptrArray
    mov ecx,arraySize
    cmp ecx,0
    ; array size != 0?
    jle L2
    ; yes: quit
    L1:
    add eax,[esi]
    ; add each integer to sum
    add esi,4
    ; point to next integer
    loop L1
    ; repeat for array size
    L2:
    pop esi
    pop ecx
    ; return sum in EAX
    leave
    ret
    8
    ; restore the stack
    ArraySum ENDP
    END
```

The code you provided is the implementation of the ArraySum() procedure in assembly language. The ArraySum() procedure calculates the sum of an array of 32-bit integers.

The ArraySum() procedure takes two parameters:

- ptrArray: A pointer to the array.
- arraySize: The size of the array.

The ArraySum() procedure works as follows:

It saves the ECX register, because it needs to use it and it does not want to overwrite the value that is in the register when the procedure is called. It sets the EAX register to 0.

This is because the EAX register will be used to store the sum of the integers in the array. It moves the pointer to the first element of the array into the ESI register. It compares the array size to 0.

If the array size is 0, the procedure exits. This is because it does not make sense to calculate the sum of an empty array.

It adds the integer at the current position in the array to the EAX register. It increments the ESI register so that it points to the next element in the array.

It repeats steps 5 and 6 until all of the integers in the array have been added to the EAX register. It restores the ECX register.

It leaves the procedure.

Here is a more detailed explanation of each step:

Step 1: Save the ECX register
The ArraySum() procedure saves the ECX register because it needs to use it and it does not want to overwrite the value that is in the register when the procedure is called.

Step 2: Set the EAX register to 0
The ArraySum() procedure sets the EAX register to 0 because it will be used to store the sum of the integers in the array.

Step 3: Move the pointer to the first element of the array into the ESI register
The ArraySum() procedure moves the pointer to the first element of the array into the ESI register. This is because the ESI register will be used to iterate through the array.

Step 4: Compare the array size to 0
The ArraySum() procedure checks the array size to make sure that it is not 0. If the array size is 0, the procedure exits. This is because it does not make sense to calculate the sum of an empty array.

Step 5: Add the integer at the current position in the array to the EAX register

The ArraySum() procedure adds the integer at the current position in the array to the EAX register. This is done using the add instruction. The add instruction adds two operands and stores the result in the first operand.

Step 6: Increment the ESI register so that it points to the next element in the array
The ArraySum() procedure increments the ESI register so that it points to the next element in the array. This is done using the inc instruction. The inc instruction increments the value of the operand by 1.

Step 7: Repeat steps 5 and 6 until all of the integers in the array have been added to the EAX register

The ArraySum() procedure repeats steps 5 and 6 until all of the integers in the array have been added to the EAX register.

This is done using the loop instruction. The loop instruction repeats a block of instructions until a specified condition is met.

In this case, the condition is that the ESI register is not equal to the value of the ptrArray parameter.

Step 8: Restore the ECX register
The ArraySum() procedure restores the ECX register.

Step 9: Leave the procedure
The ArraySum() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The ArraySum() procedure is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses a loop to iterate through the array, which is a common technique in assembly language.

=============================

Display Sum Proc

=============================

```
; DisplaySum Procedure (_display.asm)
INCLUDE Irvine32.inc
.code
    ;-----------------------------------------------------
    DisplaySum PROC
    ; Displays the sum on the console.
    ; Receives:
    ; ptrPrompt - offset of the prompt string
    ; theSum - the array sum (DWORD)
    ; Returns: nothing
    ;-----------------------------------------------------
    theSum EQU [ebp+12]
    ptrPrompt EQU [ebp+8]
    enter 0,0
    push eax
    push edx
    mov edx,ptrPrompt
    ; pointer to prompt
    call WriteString
    mov eax,theSum
    call WriteInt
    ; display EAX
    call Crlf
    pop edx
    pop eax
    leave
    ret
    8
    ; restore the stack
    DisplaySum ENDP
    END
```

The code you provided is the implementation of the DisplaySum() procedure in assembly language. The DisplaySum() procedure displays the sum of an array of 32-bit integers on the console.

The DisplaySum() procedure takes two parameters:

- **ptrPrompt:** A pointer to the prompt string.
- **theSum:** The sum of the integers in the array. The DisplaySum() procedure works as follows:

It saves the EAX and EDX registers, because it needs to use them and it does not want to overwrite the values that are in the registers when the procedure is called. It moves the pointer to the prompt string into the EDX register.

It calls the WriteString() procedure to display the prompt string on the console. It moves the sum of the integers in the array into the EAX register.

It calls the WriteInt() procedure to display the sum of the integers in the array on the console.

It calls the Crlf() procedure to move the cursor to the next line on the console. It restores the EAX and EDX registers.

It leaves the procedure.

Here is a more detailed explanation of each step:

Step 1: Save the EAX and EDX registers
The DisplaySum() procedure saves the EAX and EDX registers because it needs to use them and it does not want to overwrite the values that are in the registers when the procedure is called.

Step 2: Move the pointer to the prompt string into the EDX register
The DisplaySum() procedure moves the pointer to the prompt string into the EDX register. This is because the EDX register will be used to pass the pointer to the prompt string to the WriteString() procedure.

Step 3: Call the WriteString() procedure to display the prompt string on the console
The DisplaySum() procedure calls the WriteString() procedure to display the prompt string on the console. The WriteString() procedure is a library procedure that writes a string to the console.

Step 4: Move the sum of the integers in the array into the EAX register
The DisplaySum() procedure moves the sum of the integers in the array into the EAX register. This is because the EAX register will be used to pass the sum of the integers in the array to the WriteInt() procedure.

Step 5: Call the WriteInt() procedure to display the sum of the integers in the array on the console
The DisplaySum() procedure calls the WriteInt() procedure to display the sum of the integers in the array on the console. The WriteInt() procedure is a library procedure that writes an integer to the

console.

Step 6: Call the Crlf() procedure to move the cursor to the next line on the console
The DisplaySum() procedure calls the Crlf() procedure to move the cursor to the next line on the console. The Crlf() procedure is a library procedure that moves the cursor to the next line on the console.

Step 7: Restore the EAX and EDX registers
The DisplaySum() procedure restores the EAX and EDX registers.

Step 8: Leave the procedure
The DisplaySum() procedure leaves the procedure by using the leave instruction. The leave instruction restores the stack frame and returns from the procedure.

The DisplaySum() procedure is a good example of how to write a procedure in assembly language. The procedure is well-structured and easy to understand. The procedure also uses library procedures to perform common tasks, such as writing a string to the console and writing an integer to the console.

==============

Sum_main.asm

=============

```
; Integer Summation Program (Sum_main.asm)
; Multimodule example:
; This program inputs multiple integers from the user,
; stores them in an array, calculates the sum of the
; array, and displays the sum.
INCLUDE Irvine32.inc
INCLUDE macros.asm ; Include for INVOKE and PROTO

EXTERN PromptForIntegers:PROC
EXTERN ArraySum:PROC
EXTERN DisplaySum:PROC

; Modify Count to change the size of the array:
```

```
Count = 3

.data
prompt1 BYTE "Enter a signed integer: ",0
prompt2 BYTE "The sum of the integers is: ",0
array DWORD Count DUP(?)
sum DWORD ?

.code
main PROC
    call Clrscr
    ; PromptForIntegers(ADDR prompt1, ADDR array, Count)
    INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count

    ; sum = ArraySum(ADDR array, Count)
    INVOKE ArraySum, ADDR array, Count
    mov sum, eax

    ; DisplaySum(ADDR prompt2, sum)
    INVOKE DisplaySum, ADDR prompt2, sum
    call Crlf
    exit
main ENDP

END main
```

This code retains the same functionality as the original version but utilizes Microsoft's INVOKE and PROTO directives for calling procedures, making the code more structured and easier to read.

The code you provided is a multimodule example of an integer summation program. The program inputs multiple integers from the user, stores them in an array, calculates the sum of the array, and displays the sum.

<u>The program is divided into three modules:</u>

Sum_main.asm: This is the main module, which contains the main() procedure. The main() procedure is responsible for calling the other modules to perform the program's tasks.

promptforintegers.asm: This module contains the PromptForIntegers() procedure, which prompts the user for multiple integers and stores them in an array.

arraysum.asm: This module contains the ArraySum() procedure, which calculates the sum of the integers in an array.

display.asm: This module contains the DisplaySum() procedure, which displays the sum of the integers in an array on the console.

The Sum_main.asm module is the main module of the program. The main() procedure in this module performs the following steps:

It calls the Clrscr() procedure to clear the console screen. It calls the PromptForIntegers() procedure to prompt the user for multiple integers and store them in an array. It calls the ArraySum() procedure to calculate the sum of the integers in the array.

It calls the DisplaySum() procedure to display the sum of the integers in the array on the console. It calls the Crlf() procedure to move the cursor to the next line on the console. It calls the exit() procedure to exit the program.

The promptforintegers.asm module contains the PromptForIntegers() procedure. This procedure prompts the user for multiple integers and stores them in an array. The procedure takes the following parameters:

- **ptrPrompt:** A pointer to the prompt string.
- **ptrArray:** A pointer to the array.
- **Count:** The number of integers to prompt the user for.

The PromptForIntegers() procedure works as follows:

It iterates over the array and prompts the user for each integer. It stores the integer that the user enters in the array.

It repeats steps 1 and 2 until all of the integers have been entered. The arraysum.asm module contains the ArraySum() procedure. This procedure calculates the sum of the integers in an array. The procedure takes the following parameters:

- ptrArray: A pointer to the array.
- Count: The number of integers in the array.

The ArraySum() procedure works as follows:

It initializes the sum to 0.

It iterates over the array and adds each integer to the sum. It returns the sum. The display.asm module contains the DisplaySum() procedure.

This procedure displays the sum of the integers in an array on the console. The procedure takes the following parameters:

- **ptrPrompt:** A pointer to the prompt string.
- **theSum:** The sum of the integers in the array.

The DisplaySum() procedure works as follows:

It displays the prompt string on the console. It displays the sum of the integers in the array on the console. It moves the cursor to the next line on the console.

The integer summation program is a good example of how to use multiple modules to write a program. By dividing the program into modules, we can make the program more modular, reusable, and maintainable.

==

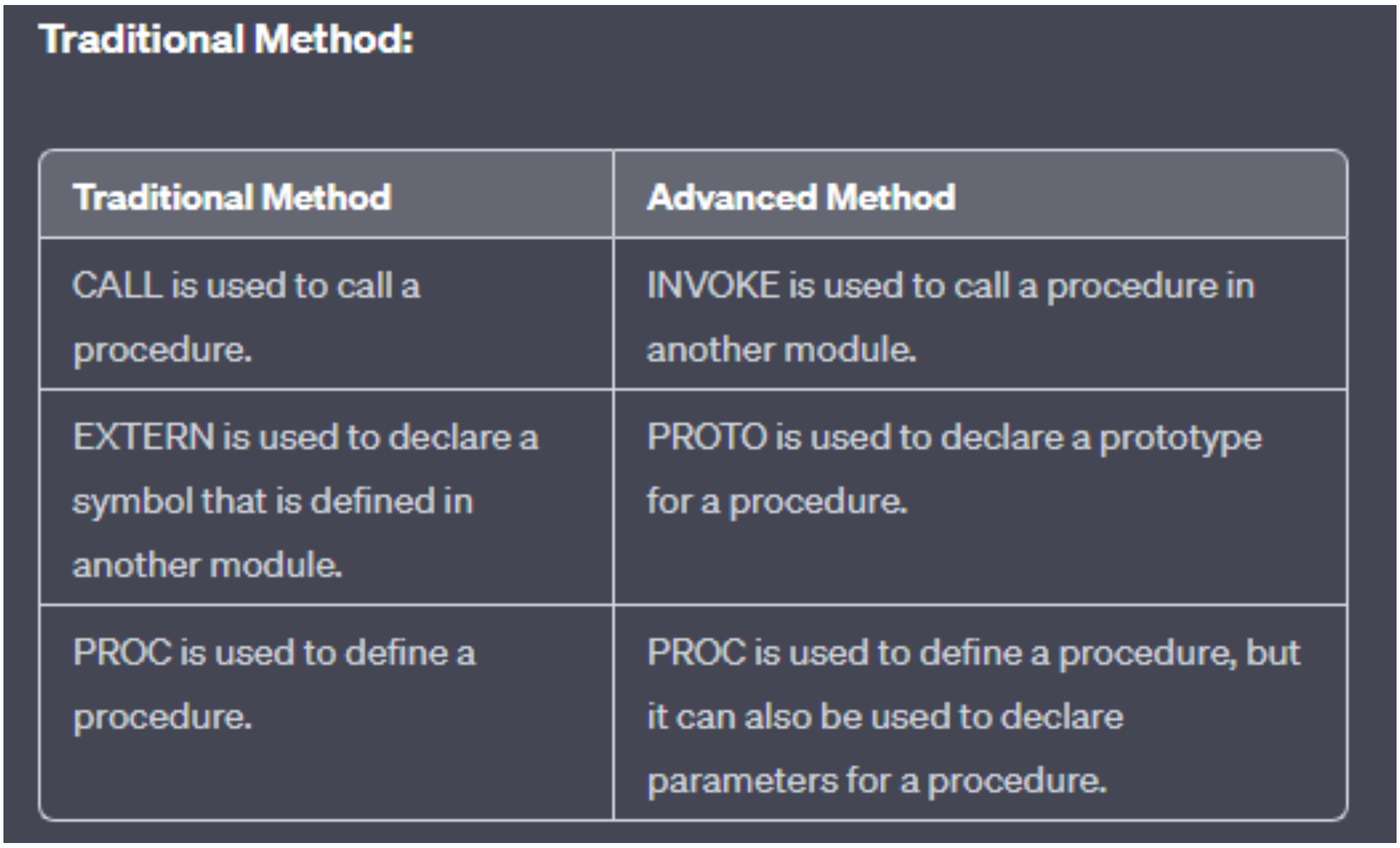
Creating Modules using INVOKE and PROTO

==

Creating the Modules Using INVOKE and PROTO section are the use of the INVOKE, PROTO, and PROC directives. These directives are used to create multimodule programs in 32-bit mode.

The INVOKE directive is used to call a procedure in another module. The PROTO directive is used to declare a prototype for a procedure. The PROC directive is used to define a procedure.

The following table shows the differences between the traditional use of CALL and EXTERN and the use of INVOKE, PROTO, and PROC:



Traditional Method:

Traditional Method	Advanced Method
CALL is used to call a procedure.	INVOKE is used to call a procedure in another module.
EXTERN is used to declare a symbol that is defined in another module.	PROTO is used to declare a prototype for a procedure.
PROC is used to define a procedure.	PROC is used to define a procedure, but it can also be used to declare parameters for a procedure.

The **main advantage of using the INVOKE, PROTO, and PROC directives** is that they allow the assembler to match up the argument lists passed by INVOKE to the corresponding parameter lists declared by PROC. This helps ensure that the program is correct and that it does not crash.

Example Using INVOKE, PROTO, and PROC:

```
; Include the necessary include file
INCLUDE sum.inc

; Define data and code sections
.data
Count = 3
prompt1 BYTE "Enter a signed integer: ",0
prompt2 BYTE "The sum of the integers is: ",0
array DWORD Count DUP(?)
sum DWORD ?

.code
main PROC
    call Clrscr

    ; Call PromptForIntegers using INVOKE with argument lists
    INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count

    ; Call ArraySum using INVOKE with argument lists
    INVOKE ArraySum, ADDR array, Count
    mov sum, eax

    ; Call DisplaySum using INVOKE with argument lists
    INVOKE DisplaySum, ADDR prompt2, sum

    call Crlf
    exit
main ENDP
```

These are all the functions using advanced methods:

```
; sum.inc
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
    ptrPrompt:PTR BYTE,
    ptrArray:PTR DWORD,
    arraySize:DWORD

ArraySum PROTO,
    ptrArray:PTR DWORD,
    arraySize:DWORD

DisplaySum PROTO,
    ptrPrompt:PTR BYTE,
    theSum:DWORD
```

```
; prompt.asm
INCLUDE sum.inc

.code
PromptForIntegers PROC,
    ptrPrompt:PTR BYTE,
    ptrArray:PTR DWORD,
    arraySize:DWORD

    pushad
    mov ecx, arraySize
    cmp ecx, 0
    jle L2
    mov edx, ptrPrompt
    mov esi, ptrArray
L1:
    call WriteString
    call ReadInt
    call Crlf
    mov [esi], eax
    add esi, 4
    loop L1
L2:
    popad
    ret
PromptForIntegers ENDP
END
```

```
; arraysum.asm
INCLUDE sum.inc

.code
ArraySum PROC,
    ptrArray:PTR DWORD,
    arraySize:DWORD

    push ecx
    mov eax, 0
    mov esi, ptrArray
    mov ecx, arraySize
    cmp ecx, 0
    jle L2
L1:
    add eax, [esi]
    add esi, 4
    loop L1
L2:
    pop ecx
    ret
ArraySum ENDP
END
```

```
; arraysum.asm
INCLUDE sum.inc

.code
ArraySum PROC,
    ptrArray:PTR DWORD,
    arraySize:DWORD

    push ecx
    mov eax, 0
    mov esi, ptrArray
    mov ecx, arraySize
    cmp ecx, 0
    jle L2
L1:
    add eax, [esi]
    add esi, 4
    loop L1
L2:
    pop ecx
    ret
ArraySum ENDP
END
```

```
; display.asm
INCLUDE sum.inc

.code
DisplaySum PROC,
    ptrPrompt:PTR BYTE,
    theSum:DWORD

    push eax
    push edx
    mov edx, ptrPrompt
    call WriteString
    mov eax, theSum
    call WriteInt
    call Crlf
    pop edx
    pop eax
    ret
DisplaySum ENDP
END
```

```
; sum_main.asm
INCLUDE sum.inc

Count = 3

.data
    prompt1 BYTE "Enter a signed integer: ", 0
    prompt2 BYTE "The sum of the integers is: ", 0
    array DWORD Count DUP(?)
    sum DWORD ?
.code
    main PROC
        call Clrscr
        INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
        INVOKE ArraySum, ADDR array, Count
        mov sum, eax
        INVOKE DisplaySum, ADDR prompt2, sum
        call Crlf
        exit
    main ENDP
    END
```

Here is a summary of the two ways to create multimodule programs:

Traditional method:

Use the EXTERN directive to declare symbols that are defined in another module. Use the CALL directive to call procedures in other modules.

Advanced method:

Use the PROTO directive to declare prototypes for procedures in other modules. Use the INVOKE directive to call procedures in other modules. Use the PROC directive to define procedures, and declare parameters for procedures. The advanced method is simpler to use and more efficient, but it is only available in 32-bit mode.

Conclusion:

The advanced method is the preferred method for creating multimodule programs in 32-bit mode. It is simpler to use and more efficient than the traditional method. However, the traditional method is still supported, and it may be necessary for some programs.