INVOKE, ADDR, PROC, and PROTO

The INVOKE, PROC, and PROTO directives provide powerful tools for defining and calling procedures in 32-bit mode.

They are more convenient to use than the traditional CALL and PROC directives, but they mask the underlying structure of the runtime stack.

In such cases, the **PROTO directive** helps the assembler to validate procedure calls by checking argument lists against procedure declarations. This can help to prevent errors and make programs more robust.

Advanced procedure directives are more convenient to use than traditional CALL and PROC directives, but they mask the underlying structure of the runtime stack.

It is important to develop a detailed understanding of the low-level mechanics involved in subroutine calls before using advanced procedure directives.

Advanced procedure directives can be used to improve program readability and maintainability, especially when programs execute procedure calls across module boundaries.

The PROTO directive helps the assembler to validate procedure calls by checking argument lists against procedure declarations.
Recommendation:

If you are new to assembly language, it is recommended that you start by learning the traditional CALL and PROC directives.

Once you have a good understanding of how subroutine calls work, you can then consider using advanced procedure directives to improve your code.

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

INVOKE Directive

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

The INVOKE directive is a powerful tool for calling procedures in 32-bit mode. It allows you to pass multiple arguments to a procedure using a single line of code.

The general syntax of the INVOKE directive is as follows:

```
INVOKE procedureName [, argumentList]
```

procedureName is the name of the procedure to be called.
argumentList is an optional comma-delimited list of arguments passed to the procedure.

Arguments to INVOKE

Arguments to INVOKE can be any valid expression, including:

- **Immediate values (e.g., 10, 3000h, OFFSET myList).**
- **Integer expressions (e.g., (1020), COUNT).**
- **Variables (e.g., myList, array, myWord, myDword).**
- **Address expressions (e.g., [myList + 2], [ebx+ esi]).**
- **Registers (e.g., eax, bl, edi).**

Arguments to INVOKE are pushed onto the stack in the reverse order that they are specified in the INVOKE statement.

The following example shows how to use the INVOKE directive to call a procedure named DumpArray():

```
INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

This statement will push the following values onto the stack:
The address of the array
The length of the array
The size of the array elements
The DumpArray() procedure will then be called with these arguments.

This statement is equivalent to the following code using the CALL instruction:

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpArray
```

The INVOKE directive can handle almost any number of arguments, and individual arguments can appear on separate source code lines. This can be useful for documenting complex INVOKE statements or for breaking up long argument lists.

The following example shows an INVOKE statement with arguments on separate source code lines:

```
INVOKE DumpArray,
; displays an array
; points to the array
OFFSET array,
; the array length
LENGTHOF array,
; array component size
TYPE array
```

Which form you choose is a matter of personal preference. Some programmers prefer to document their code extensively, while others prefer to keep their code as concise as possible.

Important Considerations

When passing arguments to INVOKE that are smaller than 32 bits, the assembler may overwrite the EAX and EDX registers when it widens the arguments before pushing them onto the stack.

To avoid this behavior, you can either pass 32-bit arguments to INVOKE or save and restore the EAX and EDX registers before and after the procedure call.

The INVOKE directive is only available in 32-bit mode.

The INVOKE directive is a powerful tool for calling procedures in 32-bit mode.

It allows you to pass multiple arguments to a procedure using a single line of code.

However, it is important to be aware of the potential for overwriting the EAX and EDX registers when passing small arguments to INVOKE.

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

ADDR Operator

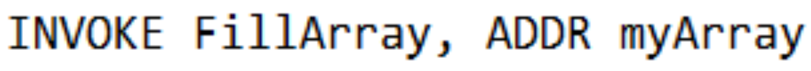
=================

The ADDR operator is a powerful tool for passing pointer arguments to procedures using INVOKE. It is only available in 32-bit mode.

The ADDR operator takes a single operand, which must be an assembly time constant. This means that the operand must be known at compile time, and cannot be a variable or expression that is evaluated at runtime.

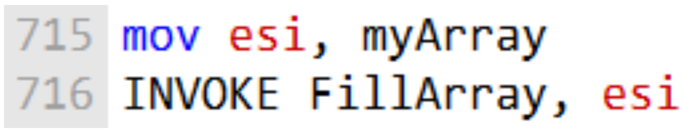
The ADDR operator returns the address of the operand. This address can then be passed to a procedure using INVOKE.

The following example shows how to use the ADDR operator to pass the address of an array to a procedure named FillArray():



```
INVOKE FillArray, ADDR myArray
```

This statement is equivalent to the following code:



```
mov esi, myArray
INVOKE FillArray, esi
```

However, the first form is more concise and readable.

The ADDR operator can only be used with the INVOKE directive.

It is not valid to use the ADDR operator with other instructions, such as MOV or CALL.

The ADDR operator can only be used to pass the address of an assembly time constant.

It is not valid to pass the address of a variable or expression that is evaluated at runtime.

The following code shows how to use the ADDR operator to call a procedure named Swap() and pass it the addresses of the first two elements in an array of doublewords:

```
.data
    Array DWORD 20 DUP(?)
.code
    ...
    INVOKE Swap,
    ADDR Array,
    ADDR [Array+4]
```

The assembler will generate the following code:

```
push
OFFSET Array+4
push
OFFSET Array
call
Swap
```

The ADDR operator is a powerful tool for passing pointer arguments to procedures using INVOKE. It allows you to write more concise and readable code.

The ADDR operator can also be used to pass the address of a function to another function. This can be useful for implementing callback functions.

For example, the following code shows how to define a function named PrintArray() that prints the elements of an array to the console:

PrintArray() that prints the elements of an array to the console:

```
.code
PrintArray PROC Near
...
ENDP
```

The following code shows how to pass the address of the PrintArray() function to a function named DoSomething():

```
.code
    DoSomething PROC Near
    INVOKE PrintArray, ADDR PrintArray
    ENDP
```

When the DoSomething() function is called, it will call the PrintArray() function to print the elements of an array to the console.

*****=================*****

PROC Operator

*****=================*****

The PROC directive is used to define a procedure in 32-bit mode. It has the following syntax:

```
label PROC [attributes] [USES reglist], parameter_list
```

• **label** is a user-defined label that follows the rules for identifiers.
• **attributes** is a list of optional attributes that can be used to control the behavior of the procedure. These attributes are:
• **distance:** Specifies whether the procedure is near or far.

- **langtype:** Specifies the calling convention (parameter passing convention) to use for the procedure.
- **visibility:** Specifies the visibility of the procedure to other modules.
- **prologuearg:** Specifies arguments affecting generation of prologue and epilogue code.
- **parameter_list** is a list of optional parameters that can be passed to the procedure.

Parameters

Parameters can be of any type, including bytes, words, doublewords, floating-point numbers, and pointers. To declare a parameter, you use the following syntax:

```
paramName:type
```

- **paramName** is the name of the parameter.
- **type** is the type of the parameter.

For example, the following procedure declares two parameters, val1 and val2, both of which are doublewords:

```
AddTwo PROC,
val1:DWORD,
val2:DWORD
```

USES

The USES clause is an optional clause that can be used to specify which registers the procedure will need to use. This can be useful for optimizing the procedure's code.

For example, the following procedure declares that it will need to use the EAX and EBX registers:

```
Read_File PROC USES eax ebx,
```

The following example shows a simple procedure named AddTwo():

```
AddTwo PROC
    val1: DWORD   ; Define a DWORD parameter named val1.
    val2: DWORD   ; Define another DWORD parameter named val2.
    mov eax, val1 ; Move the value of val1 into the EAX register.
    add eax, val2 ; Add the value of val2 to EAX.
    ret           ; Return from the procedure, effectively returning the result in EAX.
AddTwo ENDP
```

This procedure takes two doublewords as parameters and returns their sum.

The following shows the assembly code generated by MASM when assembling the AddTwo() procedure:

```
AddTwo PROC
    push ebp        ; Save the current base pointer (BP).
    mov ebp, esp    ; Set up a new base pointer, making ESP the stack frame pointer.

    mov eax, dword ptr [ebp+8]   ; Load the first parameter from the stack into EAX.
    add eax, dword ptr [ebp+0Ch] ; Add the second parameter from the stack to EAX.

    leave  ; Release the current stack frame.
    ret    ; Return from the procedure, effectively returning the result in EAX.

    8      ; Indication of the number of bytes used by parameters. (Not part of the actual code.)

AddTwo ENDP
```

The first two lines of the generated code push the EBP register onto the stack and move the stack pointer to EBP. This is done to create a stack frame for the procedure.

The next two lines move the parameters from the stack to the EAX and EDX registers.

The next line adds the two parameters together in the EAX register.

The next two lines restore the EBP register and return from the procedure.

The constant at the end of the procedure is the size of the procedure's stack frame. This value is used by the RET instruction to pop the correct number of bytes off the stack when returning from the procedure.

The PROC directive is a powerful tool for defining procedures in 32-bit mode. It allows you to create procedures with named parameters and to control the behavior of the procedure's stack frame.

Here's a detailed explanation of the code:

push ebp: This instruction saves the current base pointer (BP) by pushing it onto the stack. This is a common practice to establish a proper stack frame for the procedure.

mov ebp, esp: The ebp register is set to the current value of esp, establishing a new stack frame for this procedure. This step aligns the base pointer with the current top of the stack (ESP) and makes it easier to access function parameters and local variables.

mov eax, dword ptr [ebp+8]: This line loads the first parameter (at offset +8 from the base pointer) from the stack into the EAX register. The [ebp+8] notation indicates that the first parameter is located 8 bytes above the base pointer.

add eax, dword ptr [ebp+0Ch]: Here, the code loads the second parameter (at offset +12 from the base pointer) from the stack and adds it to the value in EAX.

leave: This instruction is often used to clean up the stack frame. It's the opposite of the enter instruction. It effectively performs the following operations:

Restores the previous value of the base pointer (EBP) from the stack. Adjusts the stack pointer (ESP) to remove the local variables and parameters of the current function. Essentially, it unwinds the stack frame to the previous state. ret: This instruction returns from the procedure, and the value in EAX becomes the return value of the function.

The **8 at the end i**s likely a comment indicating that the parameters take up 8 bytes in total (4 bytes each), which is common for two 32-bit integers.

In summary, the AddTwo procedure adds two 32-bit integers passed as parameters, and the result is returned in the EAX register. The use of the base pointer (EBP) simplifies parameter access within the stack frame.

==

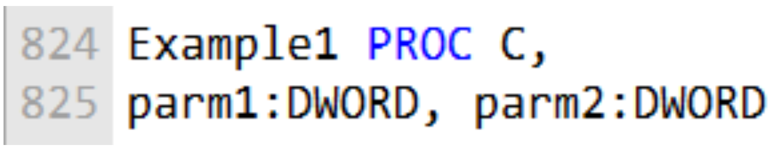
Specifying the Parameter Passing Protocol:

==

The parameter passing protocol specifies how parameters are passed to and from procedures. There are different parameter passing protocols, such as C, Pascal, and STDCALL.

To specify the parameter passing protocol for a procedure in assembly language, you can use the attributes field of the PROC directive.

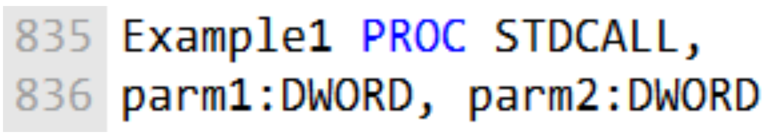
For example, the following procedure declares that it uses the C calling convention:



```
Example1 PROC C,
parm1:DWORD, parm2:DWORD
```

If you execute Example1() using the INVOKE directive, the assembler will generate code that is consistent with the C calling convention.

Similarly, the following procedure declares that it uses the STDCALL calling convention:



```
Example1 PROC STDCALL,
parm1:DWORD, parm2:DWORD
```

If you execute Example1() using the INVOKE directive, the assembler will generate code that is consistent with the STDCALL calling convention.

The following example shows how to use the PROC directive to declare a procedure with a specific parameter passing protocol:

```
Example1 PROC STDCALL,
parm1:DWORD, parm2:DWORD




.MODEL FLAT,STDCALL

; Declare a procedure with the C calling convention.
Example1 PROC C,
parm1:DWORD, parm2:DWORD

; ...

Example1 ENDP

; Declare a procedure with the STDCALL calling convention.
Example2 PROC STDCALL,
parm1:DWORD, parm2:DWORD

; ...

Example2 ENDP
```

The ability to specify the parameter passing protocol for a procedure is a powerful feature that allows you to write assembly language code that can be called from other programming languages.

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

PROTO Directive in 32-bit Mode

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

In 32-bit mode, the **PROTO directive** is used to create a prototype for an existing procedure.

A prototype declares a procedure's name and parameter list.

It allows you to call a procedure before defining it and to verify that the number and types of arguments match the procedure definition.

The syntax of the PROTO directive is as follows:

```
label PROTO [attributes] [parameter_list]
```

• **label** is the name of the procedure.
• **attributes** is an optional field that can be used to specify the parameter passing protocol for the procedure.
• **parameter_list** is an optional list of parameters that the procedure takes. Example

The following example shows how to create a prototype for a procedure named ArraySum():

```
ArraySum PROTO,
ptrArray:PTR DWORD,
; points to the array
szArray:DWORD
; array size
```

This prototype declares that the ArraySum() procedure takes two parameters: a pointer to an array of doublewords and the size of the array.

Once you have created a prototype for a procedure, you can call it using the INVOKE directive.

The INVOKE directive will verify that the number and types of arguments match the prototype before calling the procedure.

For example, the following code calls the ArraySum() procedure:

```
INVOKE ArraySum, ptrArray, szArray
```

This code will call the ArraySum() procedure with the pointer to the array ptrArray and the size of the array szArray as arguments.

Important Considerations:

Every procedure called by the INVOKE directive must have a prototype.

The prototype for a procedure must appear before the procedure is called.

The number and types of arguments in the prototype must match the number and types of arguments in the procedure definition.

The PROTO directive is a powerful tool for writing reusable and reliable assembly language code.

It allows you to call procedures before defining them and to verify that the number and types of arguments match the procedure definition.