Assembly Time Argument Checking

The PROTO directive helps the assembler check the number and types of arguments passed to a procedure when it is called. This is called assembly time argument checking.

However, assembly time argument checking is not as precise as you would find in languages like C and C++. MASM only checks for the correct number of parameters and to a limited extent, matches argument types to parameter types.

Suppose the following prototype is declared for a procedure named Sub1():

```
879 Sub1 PROTO,
880 p1:BYTE,
881 p2:WORD,
882 p3:PTR BYTE
```

This prototype declares that the Sub1() procedure takes three parameters: a byte, a word, and a pointer to a byte.

The following is a valid call to Sub1():

```
INVOKE Sub1, byte_1, word_1, ADDR byte_1
```

The assembler will generate the following code for this INVOKE statement:

```
890 push 404000h ; Push the pointer to byte_1 onto the stack.
891 sub esp, 2  ; Reserve 2 bytes on the stack for padding.
892 push word ptr ds:[00404001h] ; Push the value of word_1 onto the stack.
893 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.
894 push eax ; Push the value from EAX onto the stack.
895 call 00401071 ; Call the function at address 00401071.
```

The assembler pads the stack with two bytes because the second argument (word_1) is a word, which is two bytes long.

Errors Detected by MASM

MASM will generate an error if an argument exceeds the size of a declared parameter. For example, the following INVOKE statement will generate an error:

```
908 INVOKE Sub1, word_1, word_2, ADDR byte_1
909 ;arg 1 error
```

MASM will also generate errors if an INVOKE statement has too few or too many arguments. For example, the following INVOKE statements will generate errors:

```
913 INVOKE Sub1, byte_1, word_2
914 ; error: too few arguments
915 INVOKE Sub1, byte_1,
916 ; error: too many arguments
917 word_2, ADDR byte_1, word_2
```

Errors Not Detected by MASM

MASM will not detect an error if an argument's type is smaller than a declared parameter. For example, the following INVOKE statement will not generate an error:

```
INVOKE Sub1, byte_1, byte_1, ADDR byte_1
```

Instead, MASM will expand the smaller argument (byte_1) to the size of the declared parameter (WORD).

In the following code generated by MASM, the second argument (byte_1) is expanded into EAX before pushing it on the stack:

```
925 push 404000h ; Push the address of byte_1 onto the stack.
926 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.
927 movzx eax, al ; Expand the value in AL into EAX.
928 push eax ; Push the value from EAX onto the stack.
929 mov al, byte ptr ds:[00404000h] ; Load the value of byte_1 into AL.
930 push eax ; Push the value from EAX onto the stack.
931 call 00401071 ; Call the function at address 00401071 (Assuming it's a function).
```

Here's a more detailed explanation:

push 404000h: This instruction pushes the pointer to byte_1 onto the stack. It's pushing an address to the stack, which may be used as a parameter for the function you're calling (at address 00401071).

sub esp, 2: This instruction subtracts 2 from the stack pointer (esp). It's used to reserve 2 bytes on the stack for padding. This padding might be needed to align the stack correctly, especially when dealing with functions or system calls that expect specific stack alignment.

push word ptr ds:[00404001h]: Here, the code pushes the value of word_1 onto the stack. It's assumed that word_1 is a 16-bit (2-byte) value. The word ptr specifies that you are dealing with a word-sized value, and it's loaded from memory address 00404001h.

mov al, byte ptr ds:[00404000h]: This instruction loads the value of byte_1 into the AL register. It's assumed that byte_1 is an 8-bit (1-byte) value, and it's loaded from memory address 00404000h.

push eax: The value from the EAX register is pushed onto the stack. This is likely done to make it available as a parameter for the function being called at address 00401071.

call 00401071: This instruction calls a function located at address 00401071. The behavior of this function depends on its implementation and the purpose it serves within your program.

Overall, this code appears to be setting up some parameters on the stack and then calling a function at address 00401071, passing these parameters. The specifics of how these parameters are used and the purpose of the function being called would require more context to fully understand.

==========

ArraySum

==========

```
952 ; ArraySum Procedure
953 ; Parameters:
954 ;   esi: Points to the array
955 ;   ecx: Size of the array
956 ; Returns:
957 ;   eax: The sum of the array
958 ArraySum PROC USES esi ecx,
959     ptrArray: PTR DWORD, ; Pointer to the array
960     szArray: DWORD       ; Array size
961
962     mov esi, ptrArray  ; Load the address of the array into esi.
963     mov ecx, szArray   ; Load the size of the array into ecx.
964     mov eax, 0         ; Initialize the sum to zero.
965
966     cmp ecx, 0          ; Check if the array size is zero.
967     je L2              ; If yes, quit.
968
969 L1:
970     add eax, [esi]     ; Add the value at esi to the sum in eax.
971     add esi, 4         ; Move to the next integer in the array (4 bytes forward).
972     loop L1            ; Repeat for the remaining array size.
973
974 L2:
975     ret                ; Return with the sum in EAX.
976
977 ArraySum ENDP
```

The ArraySum() procedure takes two parameters: a pointer to an array of doublewords and the size of the array. The procedure uses the ESI and ECX registers to store the address of the array and the size of the array, respectively.

The procedure begins by setting the EAX register to zero. This will be the sum of the array elements. Then, the procedure checks the size of the array. If the size is zero, the procedure simply returns. Otherwise, the procedure enters a loop.

In the loop, the procedure adds the value at the current address in the array to the EAX register. Then, the procedure increments the ESI register to point to the next element in the array. The loop repeats until all of the elements in the array have been added.

After the loop has finished, the sum of the array elements is stored

in the EAX register. The procedure then returns.

Here is an example of how to call the ArraySum() procedure:

```
.data
    array DWORD 10000h, 20000h, 30000h, 40000h, 50000h
    theSum DWORD ?

.code
main PROC
    INVOKE ArraySum, ADDR array, LENGTHOF array
    ; Call the ArraySum procedure, passing the address of the array and the number of elements.

    mov theSum, eax
    ; Store the sum returned by ArraySum in theSum.

    ; Your program logic can continue here, using the calculated sum.

main ENDP
```

The INVOKE statement calls the ArraySum() procedure with the address of the array variable and the number of elements in the array variable as arguments.

The LENGTHOF operator is used to calculate the number of elements in the array variable.

After the ArraySum() procedure has returned, the sum of the array elements is stored in the theSum variable.

The ArraySum() example is a good example of how to use the PROC directive to declare stack parameters and how to use the INVOKE directive to call procedures with stack parameters.

```
.data
    array DWORD 10000h, 20000h, 30000h, 40000h, 50000h
    theSum DWORD ?

.code
    ; ArraySum Procedure
    ArraySum PROC USES esi ecx,
        ptrArray: PTR DWORD, ; Pointer to the array
        szArray: DWORD      ; Array size
        mov esi, ptrArray  ; Load the address of the array into esi.
        mov ecx, szArray  ; Load the size of the array into ecx.
        mov eax, 0        ; Initialize the sum to zero.
        cmp ecx, 0         ; Check if the array size is zero.
        je L2             ; If yes, quit.
    L1:
        add eax, [esi]    ; Add the value at esi to the sum in eax.
        add esi, 4        ; Move to the next integer in the array (4 bytes forward).
        loop L1           ; Repeat for the remaining array size.
    L2:
        ret               ; Return with the sum in EAX.
    ArraySum ENDP
    main PROC
        INVOKE ArraySum, ADDR array, LENGTHOF array
        ; Call the ArraySum procedure, passing the address of the array and the number of elements.
        mov theSum, eax
        ; Store the sum returned by ArraySum in theSum.
        ; Your program logic can continue here, using the calculated sum.
    main ENDP
```

In the .data section:

• An array named array is defined with five DWORD (32-bit) elements and initial values.
• A DWORD variable named theSum is declared with a question mark to indicate that it's uninitialized.

In the .code section:

• The ArraySum procedure is defined to calculate the sum of an array of DWORDs. It expects two parameters:
• **ptrArray:** A pointer to the array.
• **szArray:** The size (number of elements) of the array. Inside ArraySum:
• **esi** is used to hold the address of the array.
• **ecx** stores the size of the array.
• **eax** is initialized to zero and used to accumulate the sum.
• The code checks if the array size is zero. If it is, it immediately jumps to L2, effectively quitting the procedure.
• In L1, it adds the value at the address pointed by esi to the sum in eax, increments esi by 4 to move to the next DWORD in the array, and repeats this process for the entire array size using the loop instruction.

• Finally, in L2, it returns with the sum stored in eax.

The main procedure:

• Calls the ArraySum procedure using the INVOKE directive and passes the address of the array and the number of elements (LENGTHOF array) as parameters.
• It stores the result (the sum) returned by ArraySum in the theSum variable.
• After this code, your program logic can continue, making use of the calculated sum stored in theSum.
• This code efficiently calculates the sum of the elements in the array and stores it in theSum.

Parameter Classifications:

In the context of procedure parameters, these parameters can be classified based on the direction of data transfer between the calling program and the called procedure:

Here is a simpler explanation of input and output parameters in assembly language:

Input parameters are passed to a procedure from the calling program. The procedure can use the data, but it cannot change it. This means that when the procedure returns, the data in the calling program will be the same as it was before the procedure was called. Input parameters are typically used when the procedure needs data to operate on, but does not need to return any data.

Output parameters are used to return data from a procedure to the calling program. The procedure can change the data in the output parameter, and the calling program will see the change after the procedure returns. Output parameters are typically used when the procedure needs to return data to the calling program, such as the result of a calculation.

Here is an example of an input parameter:

```
.data
    buffer BYTE 80 DUP(?)
    inputHandle DWORD ?
.code
    INVOKE ReadConsole, inputHandle, ADDR buffer
    ; ReadConsole is expected to store user input in the 'buffer' variable.
```

and

```
procedure add_two_numbers(x: DWORD, y: DWORD): DWORD
  ; ...
  add eax, x
  add eax, y
  ret
endp

; Calling the procedure
mov eax, 10
mov ebx, 20
call add_two_numbers
mov ecx, eax ; ecx will now contain the value 30
```

In this example, the x and y parameters are input parameters. The procedure add_two_numbers() uses the data in these parameters to calculate the sum of the two numbers. However, the procedure does not change the values of x and y.

Here is an example of an output parameter:

```
1057 procedure get_system_time(time: PTR DWORD)
1058   ; ...
1059   mov [time], eax
1060   ret
1061 endp
1062
1063 ; Calling the procedure
1064 mov eax, OFFSET time_variable
1065 call get_system_time
1066
1067 ; The time variable will now contain the system time
```

In this example, the time parameter is an output parameter. The procedure get_system_time() uses the pointer in the time parameter to store the system time in the memory location that the pointer points to.

Input and output parameters can be used together in a procedure. For example, a procedure could take an input parameter that specifies the size of an array, and it could use an output parameter to return the sum of the elements in the array.

One example of an input/output parameter is a buffer. A buffer is a block of memory that is used to store data temporarily. A procedure might take an input/output parameter of type buffer to read data from a file and then return the data to the calling program.

The procedure could also use the buffer to modify the data and then return the modified data to the calling program.

Here is an example of how to use an input/output parameter in assembly language:

```
procedure read_file(buffer: PTR BYTE, size: DWORD): DWORD
  ; ...
  ; Read data from the file into the buffer
  ; ...
  ret
endp

; Calling the procedure
mov eax, OFFSET buffer
mov ebx, size
call read_file

; The buffer variable will now contain the data that was read from the file
```

In this example, the buffer parameter is an input/output parameter. The read_file() procedure reads data from the file into the buffer.

The read_file() procedure also returns the number of bytes that were read from the file. The calling program can use this information to determine how much data is in the buffer.

Example: Exchanging Two Integers

```
1090 include Irvine32.inc
1091
1092 Swap PROTO, pValX:PTR DWORD, pValY:PTR DWORD
1093 ; Exchange the values of two 32-bit integers
1094 ; Returns: nothing
1095 Swap PROC USES eax esi edi,
1096 pValX:PTR DWORD,
1097 ; pointer to first integer
1098 pValY:PTR DWORD
1099 ; pointer to second integer
1100 ; get pointers
1101 mov esi, pValX
1102 mov edi, pValY
1103 ; get first integer
1104 mov eax, [esi]
1105 ; exchange with second
1106 xchg eax, [edi]
1107 ; replace first integer
1108 mov [esi], eax
1109 ; PROC generates RET 8 here
1110 ret
1111 Swap ENDP
```

The Swap procedure takes two input/output parameters: pValX and pValY. These parameters contain the addresses of the two integers that need to be swapped.

The procedure begins by getting the pointers to the two integers.

Then, the procedure gets the value of the first integer and stores it in the EAX register.

Next, the procedure uses the **XCHG instruction** to exchange the values of the EAX register and the second integer.

Finally, the procedure stores the value of the EAX register in the first integer.

The Swap procedure does not return any value, so it simply ends with a RET instruction.

However, the PROC directive generates a RET 8 instruction at the end of the procedure, assuming that the STDCALL calling convention is being used.

The Swap procedure can be called from the main procedure as follows:

```
1117 ; Display the array before the exchange:
1118 mov esi, OFFSET Array
1119 mov ecx, 2
1120 ; count = 2
1121 mov ebx, TYPE Array
1122 call
1123 DumpMem
1124 ; dump the array values
1125 INVOKE Swap, ADDR Array, ADDR [Array+4]
1126 ; Display the array after the exchange:
1127 call
1128 DumpMem
```

The INVOKE statement calls the Swap procedure with the addresses of the first two elements of the Array variable as arguments. After the Swap procedure returns, the first two elements of the Array variable will be swapped.

Missing information:

The Swap procedure does not check for errors. For example, if the addresses of the two integers are not valid, the procedure will crash.

The Swap procedure is not optimized for speed. For example, the procedure could use a temporary variable to store the value of the first integer while the second integer is being swapped.

Overall, the Swap procedure is a simple example of how to use input/output parameters in assembly language.

==========

Debugging Tips

==========

Argument Size Mismatch

When passing arguments to a procedure, it is important to make sure that the arguments are the correct size.

For example, if a procedure expects a doubleword pointer, you should pass a doubleword pointer.

If you pass a smaller pointer, such as a word pointer, the procedure will not be able to access the data correctly.

Here is an example of an argument size mismatch:

```
1134 ; Swap procedure from Section 8.4.6
1135 Swap PROC, pValX:PTR DWORD, pValY:PTR DWORD
1136 ...
1137
1138 ; Incorrect call to Swap
1139 INVOKE Swap, ADDR [DoubleArray + 0], ADDR [DoubleArray + 1]
```

The Swap() procedure expects two doubleword pointers. However, the incorrect call to Swap() passes two word pointers.

This will cause the procedure to not be able to access the data correctly.

Passing the Wrong Type of Pointer

When passing arguments to a procedure, it is also important to make sure that the arguments are the correct type.

For example, if a procedure expects a doubleword pointer, you should pass a doubleword pointer. If you pass a different type of pointer, such as a byte pointer, the procedure will not be able to access the data correctly.

Here is an example of passing the wrong type of pointer:

```
1145 ; Swap procedure from Section 8.4.6
1146 Swap PROC, pValX:PTR DWORD, pValY:PTR DWORD
1147 ...
1148
1149 ; Incorrect call to Swap
1150 INVOKE Swap, ADDR [ByteArray + 0], ADDR [ByteArray + 1]
```

The Swap() procedure expects two doubleword pointers. However, the incorrect call to Swap() passes two byte pointers.

This will cause the procedure to not be able to access the data correctly.

Passing Immediate Values

You should not pass immediate values to reference parameters.

A **reference parameter** is a parameter that expects a pointer to data.

If you pass an immediate value to a reference parameter, the procedure will not be able to access the data correctly.

Here is an example of passing an immediate value to a reference parameter:

```
1156 ; Sub2 procedure
1157 Sub2 PROC, dataPtr:PTR WORD
1158 mov
1159 esi,dataPtr
1160 ; get the address
1161 mov
1162 WORD PTR [esi],0
1163 ; dereference, assign zero
1164 ret
1165 Sub2 ENDP
1166
1167 ; Incorrect call to Sub2
1168 INVOKE
1169 Sub2, 1000h
```

The Sub2() procedure expects a pointer to a word as its only parameter. However, the incorrect call to Sub2() passes an immediate value. This will cause the procedure to not be able to access the data correctly.

It is important to be careful when passing arguments to procedures in assembly language. If you make a mistake, it can cause the program to crash or produce incorrect results. Be sure to check the documentation for the procedure that you are calling to make sure that you are passing the correct type and number of arguments.