Variable Parameter List

March 8, 2003

Occasionally, you may find it helpful to declare and call procedures having variable parameter lists. When such functions are called, the number of arguments can vary. In the C language, for example, the printf and scanf functions let you pass a format string, followed by any number of variables and constants. The variables and constants are written to standard output, using the format specifications embedded in the format string. For example, let's ask printf to display the values of A, B, and C in various combinations:

```
int A = 20;
float B = 6.542f;
char C[] = "Hello";
printf( "%d %s", A, C );
printf( "%s", C );
printf( "%d  %f  %s \n", A, B, C );
```

In the format string, %d indicates that an integer will be printed, %f indicates that a float (real number) will be printed, and %s indicates that a string will be printed. Clearly, if the printf function is to be useful and flexible, it must accept any number of arguments. It turns out that we can do the same thing in assembly language.

Using VARARG

If your program uses the C, SYSCALL, or STDCALL calling convention (as does Irvine32.lib), you can declare procedures having a variable number of parameters. All you have to do is declare the last parameter in your procedures as type VARARG.

Following is a simple procedure named **AddAll** in which the first parameter will contain a count of the number of passed arguments when the pocedure is called. The second parameter, named **vals**, is a place marker for a variable number of arguments:

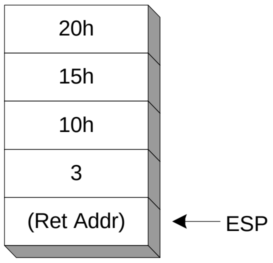
```
AddAll PROC NEAR C, argcount:DWORD, vals:VARARG
   push esi
```

```
	mov eax,0               ; clear the sum
	mov esi,0               ; initialize index register

	.WHILE  argcount > 0    ; number of arguments
	  add  eax,vals[esi]    ; vals is the first argument
	  dec  argcount         ; point to next argument
	  add  esi,TYPE DWORD
	.ENDW

	pop esi
	ret                     ; EAX contains the sum
AddAll  ENDP
```

As we study the implementation of **AddAll**, it is clear that the procedure shoulders the responsibility for locating the arguments on the stack, using a loop. The first parameter provides the loop counter, and since the arguments were pushed on the stack in reverse order, the stack looks like this when the procedure starts:



Fortunately, the last value pushed on the stack, 3, has a known location (ESP + 4). Using argument 3 as a loop counter, it's easy to move upward through the stack and reach the other three arguments.

A Windows API Example

There's a useful function in the Win32 API named **wsprintf** that formats strings, characters, and integers in a similar way to the C-language printf function. It's defined in User32.LIB, the same library featured in Section 11.2 of my book. Following is the function prototype from the MS-Windows documentation:

```
int wsprintf(
  LPTSTR lpOut,     // output buffer
  LPCTSTR lpFmt,    // format-control string
  ...               // optional arguments
);
```

The equivalent MASM prototype, located in the latest version of *SmallWin.inc*, is:

```
wsprintf PROTO NEAR C,
    lpOut:PTR BYTE,               ; output buffer
    lpFmt:PTR BYTE,               ; format-control string
    vars :VARARG                  ; optional arguments
```

As with many Win32 functions, there are two versions of this function. One has a letter "A" at the end of the function name indicates that it handles ANSI characters. Another has a letter W at the end, which handles Unicode. Typically, we provide an alternate name using TEXTEQU so the standard name can be used in our programs:

```
wsprintf TEXTEQU <wsprintfA>
```

Let's write some code that calls wsprintf and displays the resulting buffer on the console. The complete program is called *wsprintf.asm*. (Note that you will have to add User32.lib to the linker command line in the make32.bat file in the C:\Masm615 directory.)

```
.data
buffer BYTE 50 DUP(0)
fmtStr BYTE "%d, %s, 0x%X",0
intVal   DWORD -1234
unsVal   DWORD 12345678h
helloStr BYTE "Hello",0

.code
main PROC
; Format values as: signed integer, string,
; and unsigned hexadecimal:
   INVOKE wsprintf, ADDR buffer, ADDR fmtStr,
     intVal, ADDR helloStr, unsVal

; Display the buffer:
   mov edx,OFFSET buffer
   call WriteString
   call Crlf
```

The program's output looks like this:

```
-1234, Hello, 0x12345678
```

If you would like to learn about the rich set of formatting characters recognized by wsprintf, look up its documentation either on the Microsoft MSDN Web site, or in the *Platform SDK* documentation supplied with Visual C++.