AMERICAN INTERACTIVE MEDIA
A Philips/PolyGram Corporation

AIM Technical Notes

TN #44: Using Functions with Variable-Length Argument Lists in OS-9

Written by: Kirk Rader
AIM Evaluation Laboratory

September 25, 1989

OS-9 does not contain standard mechanisms for writing C code using functions that take variable-length argument lists. With special care, however, it is possible to use conventional techniques when writing C code under OS-9.

The standard I/O library for the C programming language includes functions such as **printf** and **scanf** that take variable-length argument lists. Over time, standard mechanisms for writing C code that implements such functions have developed. The standard OS-9 C compiler manual does not include any recommendations and its argument-passing scheme is more complex than the simple techniques usually provided. With some care, however, it is possible to use conventional techniques when writing C code under OS-9.

OS-9 and stdarg.h

One of the two 'standard' ways of handling variable-length argument lists is via **stdarg.h**; the other is via **varargs.h**. (The differences are in whether you grew up before or after AT&T's break up.) The conventional definition of **stdarg.h** for 68000-target compilers is:

```
#ifndef _STDARGH

#define _STDARGH

#ifndef NULL
#define NULL ((void*)0)
#endif

typedef char* va_list;

#define va_start(list,arg) \
(list = (va_list)(&arg + 1))
```

```
#define va_arg(list,type) \
(*((type*)((list+=sizeof(type))-sizeof(type))))

#define va_end(list) (list = NULL)

#endif
```

This is normally used (NOT under OS-9) as:

```
#include <stdio.h>
#include <stdarg.h>

void foo(nargs)
int nargs;
{
 if (nargs > 0) {
  int i;
  va_list arglist;
  va_start(arglist,nargs);
  for (i = 0; i < nargs; i++)
   (void)printf("Arg %d = '%s'\n",
                va_arg(arglist, char*));
  va_end(arglist);
 }
}

void main()
{
 foo(0);
 foo(1, "a");
 foo(2, "a", "b");
 foo(3, "a", "b", "c");
}
```

In the example above each argument is a character pointer. The expression **va_start** initializes a **va_list** pointer to the first optional argument after the given required argument; **va_arg** returns the current optional argument and advances the pointer to the next one; and **va_end** cleans up any register or stack-frame bashing that may be done by **va_start** and **va_arg** under some implementations. In this implemen-tation, **va_end** is not really necessary, but should be called for portability.

OS-9 Arguments

OS-9's C compiler passes the first couple of arguments (depending on their types) in registers, so the simple mechanism outlined above does not work as expected. The rule for the number arguments passed in registers and on the stack follow:

- If the first two arguments are integers or pointers, they will be in registers (D0 and D1).
- If the first argument is a floating-point number, it will be broken up into two longwords and passed in registers.

In any case, all arguments *after* those passed in registers will be pushed on the stack in the usual fashion.

The result is that the above definition of **stdarg.h** *will* work, if you take some care in handling the first few arguments as special cases. The following new definition of the function **foo** is correct for OS-9:

```
void foo(nargs, arg1, arg2)
int nargs;
char* arg1;
char* arg2;
{
 if (nargs > 0)
  (void)printf("Arg 1 = '%s'\n", arg1);
 if (nargs > 1)
  (void)printf("Arg 2 = '%s'\n", arg2);
 if (nargs > 2) {
  int i;
  va_list arglist;
  va_start(arglist,arg2);
  for (i = 3; i < nargs; i++)
   (void)printf("Arg %d = '%s'\n",
                va_arg(arglist, char*));
  va_end(list);
 }
}
```

This works because **nargs** and **arg1** are passed in registers, leaving **arg2** and any subsequent arguments on the stack. By using **va_start** on **arg2** and handling the register-passed arguments explicitly, the simple version of **stdarg.h** given above works.

Caveats

The C definition states that **char** and **short** arguments are converted to **int** before being passed to functions. So be careful to use the proper type-casting when counting arguments and accessing optional arguments.