NAME

va_arg, va_start, va_end – handle variable argument list

SYNOPSIS

#include <sys/varargs.h>

void va_start(va_list *pvar*, **void** *name***);**

type **va_arg(va_list** *pvar*, **void** *type***);**

void va_end(va_list *pvar***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

va_start( )

pvar	Pointer to variable argument list.
name	Identifier of rightmost parameter in the function definition.

va_arg( )

pvar	Pointer to variable argument list.
type	Type name of the next argument to be returned.

va_end( )

pvar	Pointer to variable argument list.

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists but do not use the **varargs** macros are inherently non-portable, as different machines use different argument-passing conventions. Routines that accept a variable argument list can use these macros to traverse the list.

va_list is the type defined for the variable used to traverse the list of arguments.

va_start( ) is called to initialize *pvar* to the beginning of the variable argument list. **va_start( )** *must* be invoked before any access to the unnamed arguments. The parameter *name* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the "**, . . .** "). If this parameter is declared with the **register** storage class or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

va_arg( ) expands to an expression that has the type and value of the next argument in the call. The parameter *pvar* must be initialized by **va_start( )**. Each invocation of **va_arg( )** modifies *pvar* so that the values of successive arguments are returned in turn. The parameter *type* is the type name of the next argument to be returned. The type name must be specified in such a way so that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a ∗ to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

The **va_end( )** macro is used to clean up. It invalidates *pvar* for use (unless **va_start( )** is invoked again).

Multiple traversals, each bracketed by a call to **va_start( )** and **va_end( )**, are possible.

EXAMPLES

The following example uses these routines to create a variable length command. This may be useful for a device which provides for a variable length command set.

ncmdbytes is the number of bytes in the command. The new command is written to **cmdp**.

```
static void
xx_write_cmd(u_char *cmdp, int ncmdbytes, ...)
{
        va_list  ap;
        int      i;

        /*
         * Write variable-length command to destination
         */
        va_start(ap, ncmdbytes);
        for (i = 0; i < ncmdbytes; i++) {
                *cmdp++ = va_arg(ap, u_char);
        }
        va_end(ap);
}
```

SEE ALSO

vcmn_err(9F), **vsprintf**(9F)

NOTES

It is up to the calling routine to specify in some manner how many arguments there are, since it is not always possible to determine the number of arguments from the stack frame.

It is non-portable to specify a second argument of **char** or **short** to **va_arg**, because arguments seen by the called function are not **char** or **short**. C converts **char** and **short** arguments to **int** before passing them to a function.