NAME

vacall − C functions called with variable arguments

SYNOPSIS

#include <vacall.h>

extern void* vacall_function;

void *function* **(va_alist***alist***)**
{
va_start_*type***(***alist***[,** *return_type***]);**
arg **= va_arg_***type***(***alist***[,** *arg_type***]);**
va_return_*type***(***alist***[[,** *return_type***],** *return_value***]);**
}

vacall_function = *&function***;**

val **= ((***return_type* **(*) ()) vacall) (***arg1***,***arg2***,...);**

DESCRIPTION

This set of macros permit a C function *function* to be called with variable arguments and to return variable return values. This is much like the **stdarg**(3) facility, but also allows the return value to be specified at run time.

Function calling conventions differ considerably on different machines, and *vacall* attempts to provide some degree of isolation from such architecture dependencies.

The function that can be called with any number and type of arguments and which will return any type of return value is **vacall**. It will do some magic and call the function stored in the variable **vacall_function**. If you want to make more than one use of *vacall*, use the *trampoline*(3) facility to store *&function* into **vacall_function** just before calling **vacall**.

Within *function*, the following macros can be used to walk through the argument list and specify a return value:

va_start_*type***(***alist***[,** *return_type***]);**
starts the walk through the argument list and specifies the return type.

arg **= va_arg_***type***(***alist***[,** *arg_type***]);**
fetches the next argument from the argument list.

va_return_*type***(***alist***[[,** *return_type***],** *return_value***]);**
ends the walk through the argument list and specifies the return value.

The *type* in **va_start_***type* and **va_return_***type* shall be one of **void**, **int**, **uint**, **long**, **ulong**, **longlong**, **ulonglong**, **double**, **struct**, **ptr** or (for ANSI C calling conventions only) **char**, **schar**, **uchar**, **short**, **ushort**, **float**, depending on the class of *return_type*.

The *type* specifiers in **va_start_***type* and **va_return_***type* must be the same. The *return_type* specifiers passed to **va_start_***type* and **va_return_***type* must be the same.

The *type* in **va_arg_***type* shall be one of **int**, **uint**, **long**, **ulong**, **longlong**, **ulonglong**, **double**, **struct**, **ptr** or (for ANSI C calling conventions only) **char**, **schar**, **uchar**, **short**, **ushort**, **float**, depending on the class of *arg_type*.

In **va_start_struct(***alist***,** *return_type***,** *splittable***);** the *splittable* flag specifies whether the struct *return_type* can be returned in registers such that every struct field fits entirely in a single register. This needs to be specified for structs of size 2*sizeof(long). For structs of size <= sizeof(long), *splittable* is ignored and assumed to be 1. For structs of size > 2*sizeof(long), *splittable* is ignored and assumed to be 0. There are some handy macros for this:
va_word_splittable_1 (*type1***)**
va_word_splittable_2 (*type1***,** *type2***)**
va_word_splittable_3 (*type1***,** *type2***,** *type3***)**
va_word_splittable_4 (*type1***,** *type2***,** *type3***,** *type4***)**

For a struct with three slots
struct { *type1 id1***;** *type2 id2***;** *type3 id3***; }**
you can specify *splittable* as **va_word_splittable_3 (***type1***,** *type2***,** *type3***)** .

NOTES

Functions which want to emulate Kernighan & Ritchie style functions (i.e., in ANSI C, functions without a typed argument list) cannot use the *type* values **char**, **schar**, **uchar**, **short**, **ushort**, **float**. As prescribed by the default K&R C expression promotions, they have to use **int** instead of **char**, **schar**, **uchar**, **short**, **ushort** and **double** instead of **float**.

The macros **va_start_longlong( )**, **va_start_ulonglong( )**, **va_return_longlong( )**, **va_return_ulonglong( )**, **va_arg_longlong( )** and **va_arg_ulonglong( )** work only if the C compiler has a working **long long** 64-bit integer type.

The struct types used in **va_start_struct( )** and **va_struct( )** must only contain (signed or unsigned) int, long, long long or pointer fields. Struct types containing (signed or unsigned) char, short, float, double or other structs are not supported.

EXAMPLE

This example, a possible implementation of **execl**(3) on top of **execv**(2) using **stdarg**(3),

```
#include <stdarg.h>
#define MAXARGS 100
/* execl is called by execl(file, arg1, arg2, ..., (char *)0); */
int execl (...)
{
  va_list ap;
  char* file;
  char* args[MAXARGS];
  int argno = 0;
  va_start (ap);
  file = va_arg(ap, char*);
  while ((args[argno] = va_arg(ap, char*)) != (char *)0)
    argno++;
  va_end (ap);
  return execv(file, args);
}
```

looks like this using **vacall**(3):

```
#include <vacall.h>
#define MAXARGS 100
/* execl is called by vacall(file, arg1, arg2, ..., (char *)0); */
void execl (va_alist ap)
{
  char* file;
  char* args[MAXARGS];
  int argno = 0;
  int retval;
  va_start_int (ap);
  file = va_arg_ptr(ap, char*);
  while ((args[argno] = va_arg_ptr(ap, char*)) != (char *)0)
    argno++;
  retval = execv(file, args);
  va_return_int (ap, retval);
}
vacall_function = &execl;
```

SEE ALSO

stdarg(3), **trampoline**(3), **callback**(3).

BUGS

The current implementations have been tested on a selection of common cases but there are probably still many bugs.

There are typically built-in limits on the size of the argument-list, which may also include the size of any structure arguments.

The decision whether a struct is to be returned in registers or in memory considers only the struct's size and alignment. This is inaccurate: for example, gcc on m68k-next returns **struct { char a,b,c; }** in registers and **struct { char a[3]; }** in memory, although both types have the same size and the same alignment.

The argument list can only be walked once.

The use of the global variable **vacall_function** is not reentrant. This is fixed in the **callback**(3) package.

PORTING

Knowledge about argument passing conventions can be found in the gcc source, file gcc-2.6.3/config/*cpu*/*cpu*.h, section "Stack layout; function entry, exit and calling."

The implementation of varargs for gcc can be found in the gcc source, files gcc-2.6.3/ginclude/va*.h.

gcc's __builtin_saveregs() function is defined in the gcc source, file gcc-2.6.3/libgcc2.c.

AUTHOR

Bruno Haible <bruno AT clisp DOT org>

ACKNOWLEDGEMENTS

Many ideas and a lot of code were cribbed from the gcc source.