Structure and Union Arguments

As described in the data representation section, structures and unions always have the alignment of their most strictly aligned member. When passed as arguments, sizes are rounded up to the next long word size. Structure and union objects appear directly on the stack, occupying as many long words as necessary.

Figure 3-19: Structure and Union Arguments

Call	Argument	Callee
`i(1, s);`	1	`8(%fp)`
	long word 0, `s`	`12(%fp)`
	long word 1, `s`	`16(%fp)`
	...	...

Functions Returning Scalars or No Value

A function that returns an integral value places its result in `%d0`. A function that returns a pointer value places its result in `%a0`. A function that returns a floating-point value places its result in `%fp0`.

Functions that return no value put no specified value in any return register.

Just as the function prologue may save registers, the epilogue must restore those same registers before returning to the caller. To complete the example from Figure 3-16, the following function epilogue restores `%d7`, `%a5`, and `%fp2` and then returns.

Figure 3-20: Function Epilogue

```
fmovm.x    (%sp)+,%fp2
movm.l     (%sp)+,%d7/%a5
unlk       %fp
rts
```

Functions Returning Structures or Unions

As mentioned above, when a function returns a structure or union, it expects the caller to provide space for the return value and to place its address in register `%a0`. Having the caller supply the return object's space allows re-entrancy.

NOTE Structures and unions in this context have fixed sizes. The ABI does not specify how to handle variable sized objects.

A function returning a structure or union also sets `%a0` to the value it finds in `%a0`. Thus when the caller receives control again, the address of the returned object resides in register `%a0`. Both the calling and the called functions must cooperate to pass the return value successfully. Failure of either side to meet its obligations leads to undefined program behavior.

The following example assumes the return object has been copied, and its address is in register `%a5`.

Figure 3-21: Function Epilogue

```
mov.l      %a5,%a0
fmovm.x    (%sp)+,%fp2
movm.l     (%sp)+,%d7/%a5
unlk       %fp
rts
```

Operating System Interface

Virtual Address Space

Processes execute in a 32-bit virtual address space. Memory management hardware translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments, commonly called text, data, and stack. As Chapter 5 describes, dynamic linking creates more segments during execution, and a process can create additional segments for itself with system services.

Page Size

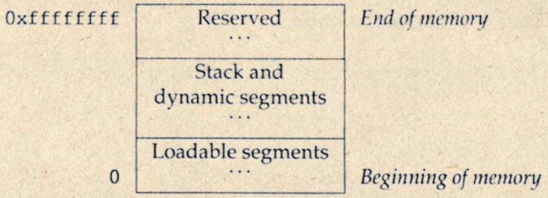
Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another, depending on the processor, memory management unit and system configuration. Allowable page sizes are 2K, 4K, or 8K. Processes may call `sysconf(BA_OS)` to determine the system's current page size.

Virtual Address Assignments

Conceptually, processes have the full 32-bit address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- A process whose size exceeds the system's available, combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amounts.

Figure 3-22: Virtual Address Configuration



CAUTION Programs that dereference null pointers are erroneous. Although such programs may appear to work on the 68000, they might fail or behave differently on other systems. To enhance portability, programmers are strongly cautioned not to rely on dereferencing null pointers.

Loadable segments
Processes' loadable segments may begin at 0. The exact addresses depend on the executable file format (see Chapters 4 and 5).

Stack and dynamic segments
A process's stack and dynamic segments reside below the reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.

Reserved
A reserved area resides at the top of virtual space.

NOTE Although application programs may begin at virtual address 0, they conventionally begin above 0x10000 (64K), leaving the initial 64K with an invalid address mapping. Processes that reference this invalid memory (for example, by dereferencing a null pointer) generate an access exception trap, as described in the "Exception Interface" section of this chapter.

As the figure shows, the system reserves the high end of virtual space, with a process's stack and dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 512 MB from the virtual address space. Thus the user virtual address range has a minimum upper bound of 0xdfffffff. Individual systems may reserve less space, increasing processes' virtual memory range. More information follows in the section "Managing the Process Stack."

Although applications may control their memory assignments, the typical arrangement follows the diagram above. Loadable segments reside at low addresses; dynamic segments occupy the higher range. When applications let the system choose addresses for dynamic segments (including shared object segments), it chooses high addresses. This leaves the "middle" of the address spectrum available for dynamic memory allocation with facilities such as malloc(BA_OS).

Managing the Process Stack

Section "Process Initialization" in this chapter describes the initial stack contents. Stack addresses can change from one system to the next—even from one process execution to the next on a single system. Processes, therefore, should *not* depend on finding their stack at a particular virtual address. The stack segment has read and write permissions.

A tunable configuration parameter controls the system maximum stack size. A process also can use setrlimit(BA_OS), to set its own maximum stack size, up to the system limit. Changes in the stack virtual address and size affect the virtual addresses for dynamic segments. Consequently, processes should *not* depend on finding their dynamic segments at particular virtual addresses. Facilities exist to let the system choose dynamic segment virtual addresses.

Coding Guidelines

Operating system facilities, such as mmap(KE_OS), allow a process to establish address mappings in two ways. First, the program can let the system choose an address. Second, the program can force the system to use an address the program supplies. This second alternative can cause application portability problems, because the requested address might not always be available. Differences in virtual address space can be particularly troublesome between different architectures, but the same problems can arise within a single architecture.

Processes' address spaces typically have three segment areas that can change size from one execution to the next: the stack [through setrlimit(BA_OS)], the data segment [through malloc(BA_OS)], and the dynamic segment area [through mmap(KE_OS)]. Consequently, an address that is available in one process execution might not be available in the next. A program that used mmap(KE_OS) to request a mapping at a specific address thus could appear to work in some environments and fail in others. For this reason, programs that wish to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility can be used properly. For example, a multiprocess application might map several files into the address space of each process and build relative pointers among the files' data. This could be done by having each process ask for a certain amount of storage at an address chosen by the system. After each process receives its own, private address from the system, it would map the desired files into memory, at specific addresses within the original area. This collection of mappings could be at different addresses in each process but their *relative* positions would be fixed. Without the ability to ask for specific addresses, the application could not build shared data structures, because the relative positions for files in each process would be unpredictable.

Processor Execution Modes

Two execution modes exist in the 68000 architecture: user and supervisor. Processes run in user mode (the least privileged). The operating system kernel runs in supervisor mode. A program executes the trap instruction to change execution modes.

NOTE The ABI does not define the implementation of individual system calls. Instead, programs shall use the system libraries that Chapter 6 describes. Programs with embedded system call trap instructions do not conform to the ABI.

Exception Interface

As the 68000 user manuals describe, the processor also changes mode to handle *exceptions*. Exceptions can be explicitly generated by a process as a result of instruction execution. The operating system defines the following correspondence between hardware exceptions and the signals specified by signal(BA_OS).

Figure 3-23: Exceptions and Signals

Exception Name	Signal
trap #1 (breakpoint trap)	SIGTRAP
external memory fault	see below
address error	SIGBUS
all floating-point exceptions	SIGFPE
illegal instruction	SIGILL
integer zero-divide	SIGFPE
privileged opcode	SIGILL
trace	SIGTRAP
chk, chk2 instruction	SIGFPE
cptrapcc, trapcc, trapv	SIGFPE
line 1010 emulator	SIGSYS
line 1111 emulator	SIGSYS
trap #2-15	SIGSYS

An external memory fault exception can generate various signals, depending on why the exception occurred.

SIGBUS or SIGEMT
The process accessed a memory location in a way disallowed by the current mapping's permissions. As an example, the process tried to store a value into a location without write permission.

SIGSEGV The process referenced a memory address for which no valid mapping existed.

Process Initialization

This section describes the machine state that exec(BA_OS) creates for "infant" processes, including argument passing, register usage, stack frame layout, etc. Programming language systems use this initial program state to establish a standard environment for their application programs. As an example, a C program begins executing at a function named main, conventionally declared in the following way.

Figure 3-24: Declaration for main

```
extern int main(int argc, char *argv[], char *envp[]);
```

Briefly, argc is a non-negative argument count; argv is an array of argument strings, with argv[argc]==0; and envp is an array of environment strings, also terminated by a null pointer.

Although this section does not describe C program initialization, it gives the information necessary to implement the call to main or to the entry point for a program in any other language.

Registers

Registers %d0 through %d7, %a0 through %a6, and all MC68040 or MC68881/2 FPCP floating-point data registers have unspecified values at process entry. The floating-point control registers are set to provide IEEE default behavior. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should *not* rely on the operating system to set all registers to 0. See "Process Stack" below for information about the initial values of the stack registers.

As the architecture defines, the status register controls and monitors the processor. Application programs cannot access the entire status register directly; they run in the processor's *user mode*, and the instructions to access the entire status register are privileged. Nonetheless, a program can access the condition code register, which initially has the following value.

Figure 3-25: Condition Code Register Fields

Field	Value	Note
XNZVC	unspecified	Condition codes unspecified

Process Stack

When a process receives control, its stack holds the arguments and environment from exec(BA_OS).

Figure 3-26: Initial Process Stack

	Unspecified . . .	*High Addresses*
	Information block, including argument strings environment strings auxiliary information . . . size varies	
	Unspecified . . .	
	Null auxiliary vector entry	
	Auxiliary vector . . . 2-long word entries	
	0 long word	
	Environment pointers . . . one long word each	
	0 long word	
4(%sp)	Argument pointers . . . *Argument count* long words	
0(%sp)	Argument count	*Low Addresses*
	Undefined . . .	

Every process has a stack, but the system defines *no* fixed stack address. Furthermore, a program's stack address can change from one system to another—even from one process invocation to another. Thus the process initialization code must use the address in %sp.

Argument strings, environment strings, and auxiliary information appear in no specific order within the information block; the system makes no guarantees about their arrangement. The system may leave an unspecified amount of memory between the null auxiliary vector entry and the start of the information block.

Whereas the argument and environment vectors transmit information from one application program to another, the auxiliary vector conveys information from the operating system to the program. This vector is an array of the following structures, interpreted according to the a_type member.

Figure 3-27: Auxiliary Vector

```
typedef     struct
{
            int       a_type;
            union {
                      long      a_val;
                      void      *a_ptr;
                      void      (*a_fcn)();
            } a_un;
} auxv_t;
```

Figure 3-28: Auxiliary Vector Types, a_type

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val

Figure 3-28: Auxiliary Vector Types, a_type **(continued)**

Name	Value	a_un
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr

AT_NULL The auxiliary vector has no fixed length; instead its last entry's a_type member has this value.

AT_IGNORE This type indicates the entry has no meaning. The corresponding value of a_un is undefined.

AT_EXECFD As Chapter 5 describes, exec(BA_OS) may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.

AT_PHDR Under some conditions, the system creates the memory image of the application program before passing control to the interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See Chapter 5 in both the System V ABI and the processor supplement for more information about the program header table.

AT_PHENT The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.

AT_PHNUM The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.

AT_PAGESZ If present, this entry's a_val member gives the system page size, in bytes. The same information also is available through sysconf(BA_OS).

AT_BASE The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See "Program Header" in the System V ABI for more information about the base address.

AT_FLAGS If present, the a_val member of this entry holds one-bit flags. Bits with undefined semantics are set to zero. No flags are defined for the 68000.

AT_ENTRY The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

Other auxiliary vector types are reserved.

In the following example, the stack resides at an address below 0xf0000000, growing toward lower addresses. The process receives three arguments.

- cp
- src
- dst

It also inherits two environment strings (this example is not intended to show a fully configured execution environment).

- HOME=/home/dir
- PATH=/home/dir/bin:/usr/bin:

Its auxiliary vector holds one non-null entry, a file descriptor for the executable file.

- {AT_EXECFD, 13}

Figure 3-29: Example Process Stack

0xeffffffc	\0	c	p	\0	*High addresses*
	\0	s	r	c	
	\0	d	s	t	
oxeffffff0	/	d	i	r	
	h	o	m	e	
	M	E	=	/	
	:	\0	H	O	
oxefffffe0	/	b	i	n	
	/	u	s	r	
	b	i	n	:	
	d	i	r	/	
0xefffffd0	o	m	e	/	
	H	=	/	h	
0xefffffc8	*pad*	P	A	T	
	0				
0xefffffc0	0				
	13				
	2				Auxiliary Vector
	0				
0xefffffb0	0xefffffc9				
	0xefffffe6				Environment vector
	0				
	0xeffffff5				
0xefffffa0	0xeffffff9				
	0xefffffffd				Argument Vector
%sp, 0xefffff98	3				Argument Count
-4(%sp), 0xefffff94	Undefined				*Low addresses*

Coding Examples

This section discusses example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discuss how a program may use the machine or the operating system, and they specify what a program may and may not assume about the execution environment. The information here illustrates how operations *may* be done, not how they *must* be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does *not* prevent a program from conforming to the ABI. Two main object code models are available.

- *Absolute code*. Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process's virtual addresses.
- *Position-independent code*. Instructions under this model hold relative addresses, *not* absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

The following sections describe the differences between these models. Code sequences for the models (when different) appear together, allowing easier comparison.

NOTE Examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences nor to reproduce compiler output.

NOTE When other sections of this document show assembly language code sequences, they typically show only the absolute versions. Information in this section explains how position-independent code would alter the examples.

Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared object libraries conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change the segment images. Thus multiple processes can share a single shared object text segment, even though the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques.

- Control transfer instructions hold addresses relative to the program counter (PC). A PC-relative branch or function call computes its destination address in terms of the current program counter, *not* relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the instructions, the compiler generates code to calculate an absolute address during execution.

Because the processor architecture provides PC-relative call and branch instructions, compilers can satisfy the first condition easily.

A *global offset table* and a *procedure linkage table* provide information for address calculation. Position-independent object files (executable and shared object files) have these tables in their data segment. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual addresses as assigned for an individual process. Because data segments are private for each process, the table entries can change—unlike text segments, which multiple processes share.

Assembly language examples below show the explicit notation needed for position-independent code.

name@GOT
This expression denotes the displacement in the global offset table of the entry for the symbol *name*.

name@PLT
This expression denotes the displacement in the procedure linkage table of the entry for the symbol *name*.

name@GOTPC
This expression denotes a PC-relative reference to the global offset table entry for the symbol *name*.

name@PLTPC
This expression denotes a PC-relative reference to the procedure linkage table entry for the symbol *name*.

Position-Independent Function Prologue

This section describes the function prologue for position-independent code. A function's prologue first allocates the local stack space. Position-independent functions also set register %a5 to the global offset table's address, accessed with the symbol _GLOBAL_OFFSET_TABLE_. Because %a5 is private for each function and preserved across function calls, a function calculates its value once at the entry.

NOTE

As a reminder, this entire section contains examples. Using %a5 is a convention, not a requirement; moreover, this convention is private to a function. Not only could other registers serve the same purpose, but different functions in a program could use different registers.

NOTE

When an instruction uses _GLOBAL_OFFSET_TABLE_@GOTPC, it sees the offset between the current instruction and the global offset table as the symbol value.

Figure 3-30: Position-Independent Function Prologue

```
name:
   link.l   %fp,&-80
   movm.l   %a5,-(%sp)
   lea      (%pc,_GLOBAL_OFFSET_TABLE_@GOTPC),%a5
```

Data Objects

This discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack and frame pointers. Instead, this section describes objects with static storage duration. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Figure 3-31: Absolute Load and Store

C	Assembly
extern int src;	.global src,dst,ptr
extern int dst;	
extern int *ptr;	
ptr = &dst;	mov.l &dst,ptr
*ptr = src;	mov.l src,([ptr])

Position-independent code cannot contain absolute addresses. Referencing global symbols must be done with a base register and global offset table index (as in these examples). Alternatively, instructions that reference symbols hold the PC-relative offsets into the global offset table. Combining the offset with the PC gives the absolute address of the table entry holding the desired address.

Figure 3-32: Position-Independent Load and Store

C	Assembly
extern int src;	.global src,dst,ptr
extern int dst;	
extern int *ptr;	
ptr = &dst;	mov.l (%a5,dst@GOT),([%a5,ptr@GOT])
*ptr = src;	mov.l ([%a5,ptr@GOT]),%a4
	mov.l ([%a5,src@GOT]),(%a4)

Function Calls

Function calls in absolute code put the symbols' values—or absolute virtual addresses—into instructions. Programs use the jsr or bsr instructions to make function calls. For absolute code, the destination operand is an absolute address. Even when the code for a function resides in a shared object, the caller uses the same assembly language instruction sequence, although in that case control passes from the original call, through an indirection sequence, to the desired destination. See "Procedure Linkage Table" in Chapter 5 for more information on the indirection sequence.

Figure 3-33: Absolute Direct Function Call

C	Assembly
extern void function();	.global function
function();	jsr function

For position-independent code, the `bsr` instruction's destination operand is a PC-relative value. Typically, the destination will be an entry in the procedure linkage table mentioned above.

Figure 3-34: Position-Independent Direct Function Call

C

```
extern void function();
function();
```

Assembly

```
.global   function
bsr       function@PLTPC
```

Indirect function calls also use the `jsr` instruction.

Figure 3-35: Absolute Indirect Function Call

C

```
extern void (*ptr)();
extern void name();
ptr = name;

(*ptr)();
```

Assembly

```
.global   ptr,name

mov.l     &name,ptr

jsr       ([ptr])
```

For position-independent code, the global offset table supplies absolute addresses for all required symbols, whether the symbols name objects or functions.

Figure 3-36: Position-Independent Indirect Function Call

C

```
extern void (*ptr)();
extern void name();
ptr = name;

(*ptr)();
```

Assembly

```
.global   ptr,name

mov.l     (%a5,name@GOT),([%a5,ptr@GOT])

mov.l     ([%a5,ptr@GOT]),%a0
jsr       (%a0)
```

Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions can hold a PC-relative value with a range that covers the entire address space.

Figure 3-37: Branch Instruction, Both Models

C

```
label:
            ...
            goto label;
```

Assembly

```
.L01:
            ...
            bra.l   .L01
```

C `switch` statements provide multiway selection. When the case labels of a `switch` statement satisfy grouping constraints, the compiler implements the selection with an address table. The following examples use several simplifying conventions to hide irrelevant details:

- The selection expression resides in register `%d0`;
- case label constants begin at zero;
- case labels, `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab`, respectively.

Address table entries for absolute code contain virtual addresses; the selection code extracts an entry's value and jumps to that address. Position-independent table entries hold offsets; the selection code computes a destination's absolute address.

Figure 3-38: Absolute `switch` Code

C

```
switch (j)
{
case 0:
    ...
case 2:
    ...
case 3:
    ...
default:
    ...
}
```

Assembly

```
        cmp.l   %d0,&3
        bhi     .Ldef
        asl.l   &2,%d0
        jmp     ([%pc,%d0.l,.Ltab])
        ...
.Ltab:  .long   .Lcase0
        .long   .Ldef
        .long   .Lcase2
        .long   .Lcase3
```

Figure 3-39: Position-Independent `switch` Code

C

```
switch (j)
{
case 0:
    ...
case 2:
    ...
case 3:
    ...
default:
    ...
}
```

Assembly

```
        cmp.l   %d0,&3
        bhi     .Ldef
        mov.l   (%pc,%d0.w*4,.Ltab),%d0
        jmp     (%pc,%d0,.Ltab)




        ...
.Ltab:  .long   .Lcase0-.Ltab
        .long   .Ldef-.Ltab
        .long   .Lcase2-.Ltab
        .long   .Lcase3-.Ltab
```

C Stack Frame

Figure 3-40 shows the C stack frame organization. It conforms to the standard stack frame with designated roles for unspecified areas in the standard frame.

Figure 3-40: C Stack Frame

Base	Offset	Contents	Purpose	
	+4+4**n*	argument long word *n*	incoming arguments	*High Addresses*
		...		
	+4	argument long word 0		
SP′	(SP before call)	return address		
		unspecified	local storage and register save area	
		...		
SP	(SP after call)	variable size		
		outgoing arguments		
		stack top, unused		*Low addresses*

Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments reside on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many machines, including the 68000. Nonetheless, portable C programs should use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists (on 68000 and other machines as well).

Allocating Stack Space Dynamically

Unlike some other languages, C does not need dynamic stack allocation *within* a stack frame. Frames are allocated dynamically on the program stack, depending on program execution. The architecture supports dynamic allocation for those languages that require it, and the standard calling sequence and stack frame support it as well. Thus languages that need dynamic stack frame sizes can call C functions, and vice versa.

Figure 3-40 shows the layout of the C stack frame. The double line divides the area allocated by the compiler from the dynamically allocated memory. Dynamic space is allocated below the line as a downward growing heap whose size changes as required. Typical C functions have no space in the heap. All areas above the double line in the current frame have a known size to the compiler. Dynamic stack allocation thus takes the following steps.

1. Stack frames are long word aligned; dynamic allocation should preserve this property. Thus the program rounds (up) the desired byte count to a multiple of 4.
2. The program decreases the stack pointer by the rounded byte count, increasing its frame size. At this point, the "new" space resides just below the register save area at the bottom of the stack.

Even in the presence of signals, dynamic allocation is "safe." If a signal interrupts allocation, one of three things can happen.

- The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.
- The signal handler can execute a non-local goto, or `longjmp` [see `setjmp`(BA_LIB)]. This resets the process to a new context in a previous stack frame, automatically discarding the dynamic allocation.
- The process can terminate.

Regardless of when the signal arrives during dynamic allocation, the result is a consistent (though possibly dead) process.