Trap Manager

This chapter describes how your application can use the Trap Manager to augment or override an existing system software routine.

Although this chapter describes patching in some depth, you should rarely, if ever, find a need to use patches in an application. The primary purposes of patches, as their name suggests, are to fix problems and augment routines in ROM code.

To use this chapter, you should have some knowledge of assembly language. For information about the instruction sets of microprocessors in the Motorola MC680x0 family, see the appropriate user's manual, for example, the *MC68020 32-Bit Microprocessor User's Manual.*

This chapter describes how the Trap Manager works and then shows how you can use the Trap Manger to

- ■ check for the availability of a system software routine
- ■ alter the behavior of a system software routine

About the Trap Manager

The Trap Manager is a collection of routines that lets you add extra capabilities to system software routines.

In order to execute system software routines, system software takes advantage of the unimplemented instruction feature of the MC680x0 family of microprocessors, which are the central processing units (CPUs) used in the Macintosh family of computers.

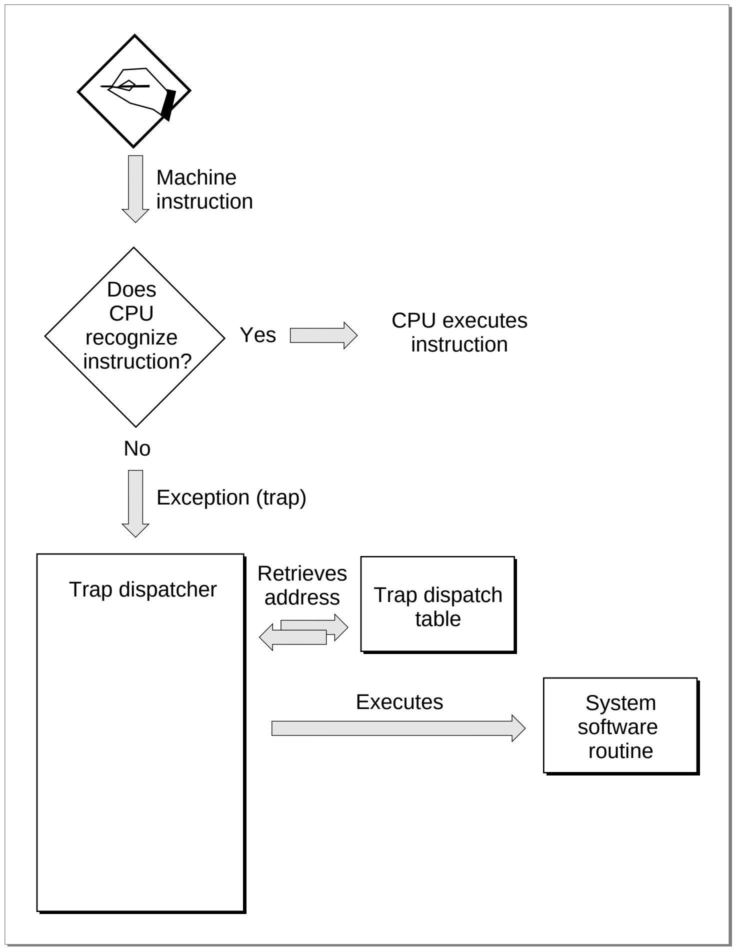
The MC680x0, like other microprocessors, executes a stream of instructions. Information encoded in an instruction indicates the operation to be performed by the microprocessor. The MC680x0 family of microprocessors recognizes a defined set of instructions. When the microprocessor encounters an instruction that it doesn't recognize, an exception is generated. An exception refers to bus errors, interrupts, and unimplemented instructions. When an exception occurs, the microprocessor suspends normal execution and transfers control to an appropriate exception handler.

In the MC680x0 family of microprocessors, all instructions starting with the hexadecimal digit $A are unimplemented instructions. These unimplemented instructions are also called ***A-line instructions***. System software uses these unimplemented A-line instructions to execute system software routines. When you call a system software routine, the call to the system software routine is translated into an A-line instruction. The MC680x0 microprocessor doesn't recognize this A-line instruction, and transfers control to an exception handler.

System software provides an exception handler, called a ***trap dispatcher***, to handle exceptions generated by A-line instructions. Whenever a MC680x0 microprocessor encounters an A-line instruction, an exception is generated, and the microprocessor transfers control to the trap dispatcher. An exception generated by an A-line instruction is called a ***trap***.

When the trap dispatcher receives the A-line instruction, it looks into a table, called a ***trap dispatch table***, to find the address of the called system software routine. After the trap dispatcher retrieves the address, it transfers control to the specified system software routine. Figure 8-1 illustrates the processing of instructions that include the A-line instructions that the microprocessor does not recognize.

Figure 8-1 How the CPU processes A-line instructions

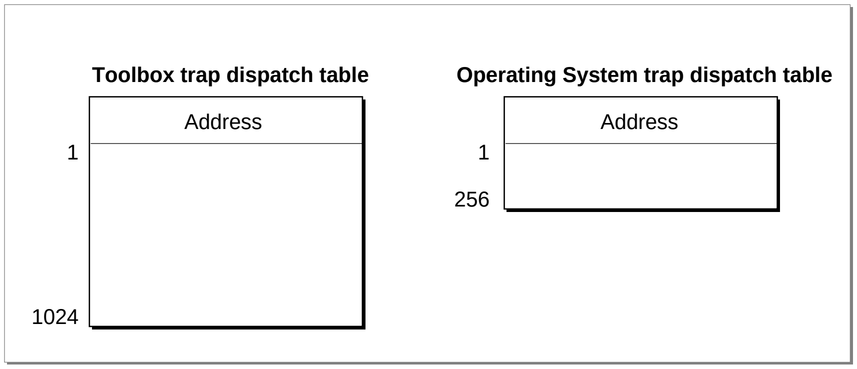


You can use the Trap Manager routines to read from and write to the two trap dispatch tables maintained by system software.

Trap Dispatch Tables

System software uses trap dispatch tables to locate the address of system software routines. System software maintains two trap dispatch tables: an Operating System trap dispatch table and a Toolbox trap dispatch table. Figure 8-2 illustrates the two trap dispatch tables.

Figure 8-2 Trap dispatch tables

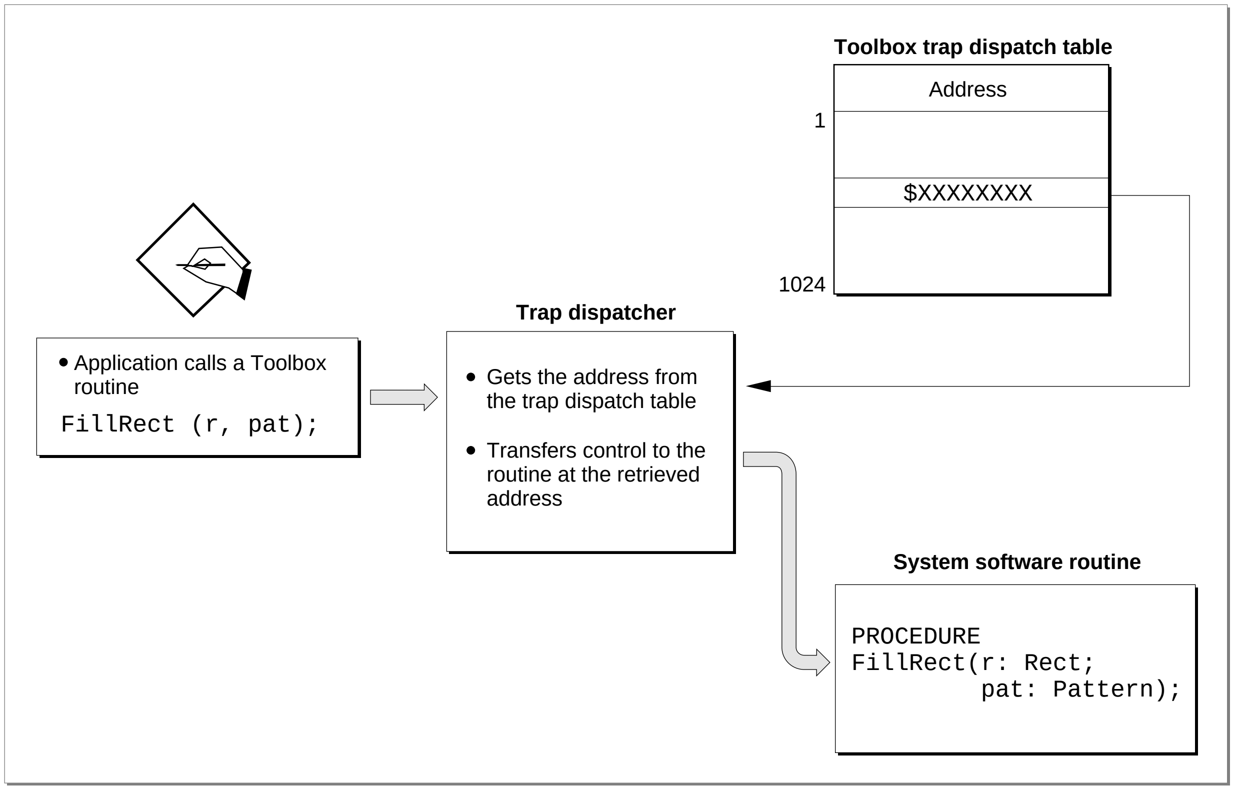

Toolbox trap dispatch table
Address
1
1024
Operating System trap dispatch table
Address
1
256

At system startup time, system software builds the trap dispatch tables and places them in RAM. The ***Operating System trap dispatch table*** contains 256 entries, and the ***Toolbox trap dispatch table*** contains 1024 entries. Each entry in the Operating System trap dispatch table contains a 32-bit address of an Operating System routine, and each entry in the Toolbox trap dispatch table contains a 32-bit address of a Toolbox routine. The system software routines can be located in either ROM or RAM.

Process for Accessing System Software Routines

As previously described, when your application calls a system software routine, an A-line instruction is sent to the microprocessor. The microprocessor does not recognize this instruction, and an exception is generated. This exception is then handled by the trap dispatcher. When the trap dispatcher receives the A-line instruction, it looks into one of the two trap dispatch tables to find the address of the called system software routine. When the trap dispatcher retrieves the address, it transfers control to the specified system software routine. For example, Figure 8-3 illustrates a call to the Toolbox procedure, `FillRect`. When the application calls the `FillRect` procedure, an exception is generated. The trap dispatcher looks into the Toolbox trap dispatch table to find the address of the `FillRect` procedure. When the address is found, the trap dispatcher transfers control to the `FillRect` procedure.

Figure 8-3 Accessing the `FillRect` procedure


Toolbox trap dispatch table
Address
1
$XXXXXXXX
1024
Trap dispatcher
• Application calls a Toolbox routine
FillRect (r, pat);
• Gets the address from the trap dispatch table
• Transfers control to the routine at the retrieved address
System software routine
PROCEDURE FillRect(r: Rect; pat: Pattern);

Note
Not all A-line instructions are defined. When the trap dispatcher receives an undefined A-line instruction, the trap dispatcher returns the address of the Toolbox procedure `Unimplemented`. When called, the `Unimplemented` procedure triggers a system error. ◆

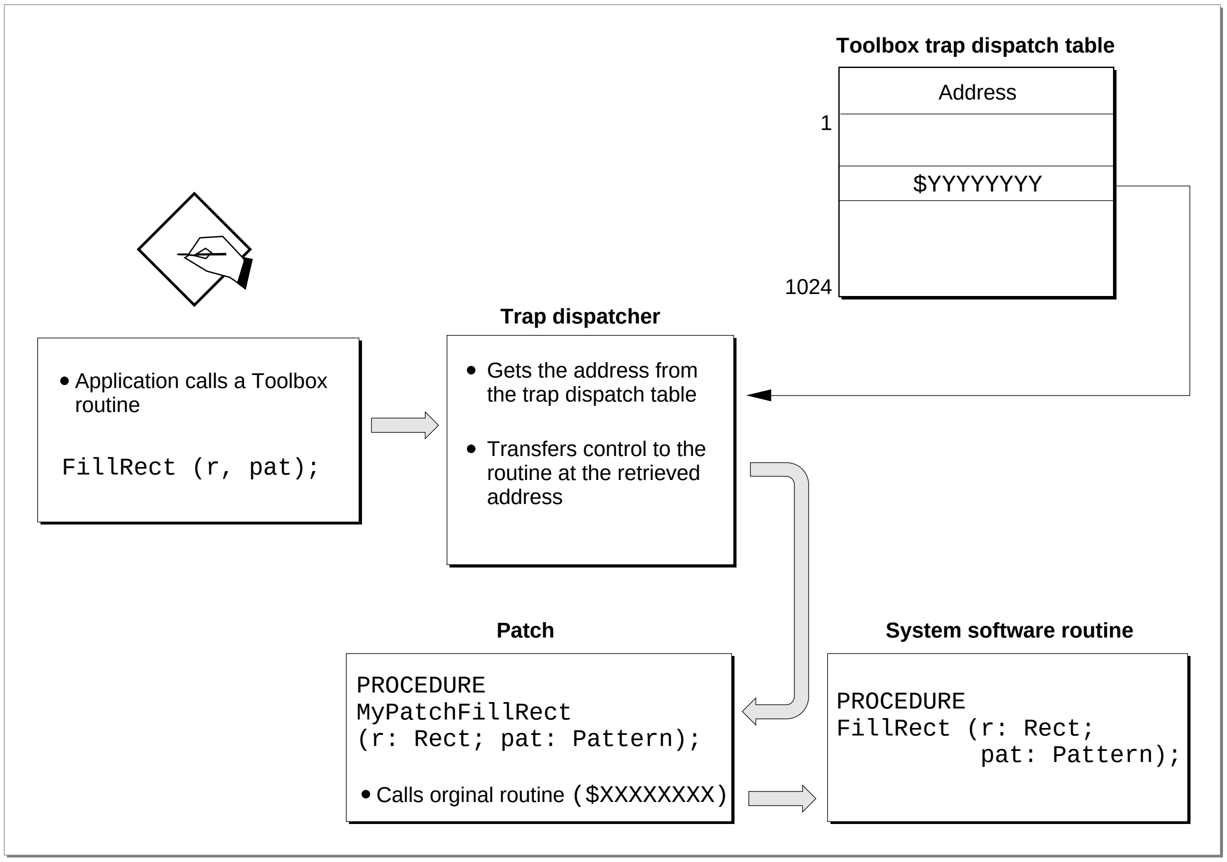
Patches and System Software Routines

You can modify the trap dispatch table so that the address that gets returned to the trap dispatcher points to a different routine instead of the intended system software routine; this is useful if you want to augment or override an existing system software routine. The routine that augment an existing system software routine is called a ***patch***. The method of augmenting or overriding a system software routine is called *patching a trap*.

For example, you can augment the `FillRect` procedure with your own procedure `MyPatchFillRect`. Figure 8-4 illustrates another call to the Toolbox procedure `FillRect`. When the application calls the `FillRect` procedure the application-defined patch `MyPatchFillRect` is executed first. After the application-defined patch `MyPatchFillRect` completes its primary action, it transfers control (through a JMP instruction) to the original `FillRect` procedure.

IMPORTANT
Although this chapter describes patching in some detail, you should avoid any unnecessary patching of the system software. One very good reason to avoid patching is that is causes a performance reduction. The performance reduction is especially substantial when your patch is executed on a PowerPC processor-based Macintosh computer, where it is necessary to switch execution environments when entering and exiting your patch code. For more information about patching PowerPC system software, see *Inside Macintosh: PowerPC System Software*. ▲

Figure 8-4 Augmenting the `FillRect` procedure with a single patch


Toolbox trap dispatch table
Address
1
$YYYYYYYY
1024
Trap dispatcher
• Gets the address from the trap dispatch table
• Transfers control to the routine at the retrieved address
• Application calls a Toolbox routine
FillRect (r, pat);
Patch
PROCEDURE
MyPatchFillRect
(r: Rect; pat: Pattern);
• Calls orginal routine ($XXXXXXXX)
System software routine
PROCEDURE
FillRect (r: Rect;
pat: Pattern);

Note
To prevent dangling patch addresses, you must ensure that your patch routine is in a locked memory block while its address is in the trap dispatch table. ◆

Daisy Chain of Patches

It is possible to patch a system software routine with more than just one patch; this is called a ***daisy chain*** of patches. Typically, you extract from the trap dispatch table the address of the routine you wish to patch, save this address, and then install your own patch routine. When your patch has completed its tasks, it should jump to the address you previously extracted from the trap dispatch table. In this way, the patches take the general form of a daisy chain. Each patch will execute in turn and jump to the next patch until the last link in the chain, which returns control to the trap dispatcher.

IMPORTANT
Although this chapter describes patching in some depth, you should rarely, if ever, find a need to use patches in an application. The primary purposes of patches, as their name suggests, are to fix problems and augment routines in ROM code. ▲

A patch can be implemented as either a head patch, tail patch, or come-from patch. These are described in the next sections.

Head Patch (Normal Patch)

A ***head patch***, also referred to as a ***normal patch***, is a routine that gets executed before the original system software routine. A head patch performs its primary action and then uses a jump instruction (`JMP`) to jump to the system software routine. Thus the head patch does not regain control after the execution of the system software routine. After the execution of the system software routine, control is transferred back to the trap dispatcher.

Tail Patch

A ***tail patch*** is a routine that gets executed before the original system software routine and regains control after the execution of the system software routine. A tail patch uses a jump-subroutine instruction (`JSR`) to transfer control to the system software routine. After the system software routine returns control to the tail patch, the tail patch returns control to the trap dispatcher.

▲ **WARNING**
You should never install tail patches in system software versions earlier than System 7. Tail patches may conflict with come-from patches, installed by Apple. ▲

Come-From Patch (Used Only by Apple)

A ***come-from patch***, also called a ***system patch***, is a type of patch used only by Apple. Come-from patches are used to replace erroneous code or to add capabilities not in ROM.

When a come-from patch is invoked, it examines the stack to determine where it was called from. If the come-from patch was invoked from a particular place in ROM (a spot where the code needs to be augmented or deleted), the come-from patch executes the modifying code. Otherwise, if the come-from patch was called from a part of the system that does not need to be augmented, it transfers control to the next routine in the daisy chain. This routine could be another patch or the system software routine.

Beginning with System 7, the addresses of come-from patches are permanently placed in the trap dispatch table at system startup time. The addresses of come-from patches are hidden and cannot be manipulated by any of the Trap Manger routines.

For example, if a system software routine has a come-from patch and if you use the Trap Manger function `NGetTrapAddress` to retrieve the address of the system software routine, you will not get the address in the trap dispatch table (which is the address of the come-from patch). `NGetTrapAddress` instead returns the address of the routine that is executed immediately after the come-from patch. This address could be the address of another patch or the system software routine.

If a system software routine has a come-from patch and if you use the Trap Manager procedure `NSetTrapAddress` to install a patch to the system software routine, the address of the patch is not written into the trap dispatch table. Instead, the `NSetTrapAddress` procedure installs the address of the patch into the last come-from patch. The patch is executed after the completion of the come-from patch.

▲ **WARNING**
In system software before System 7, if a come-from patch is invoked by a tail-patch, the come-from patch does not work correctly. The come-from patch never sees the ROM address on the stack—only the return address of the tail-patch. ▲

Patch for One Application

If you install a patch into your application heap, the patch applies only to your application. When your application is switched out, your application's heap (and patch) is swapped out. For example, if you patch `FillRect` with the patch `MyPatchFillRect`, the `MyPatchFillRect` patch is executed only when the `FillRect` procedure is called from your application.

Note
When running in System 7 or under MultiFinder in System 6, each application has its own copy of the trap dispatch tables. This ensures that an application's patches apply only when it is running and that they're discarded when the application quits. ◆

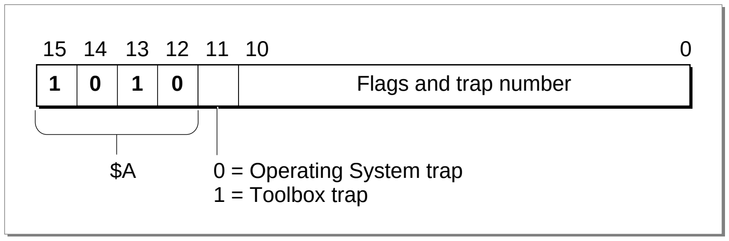
Patch for All Applications

If you install a patch from a system extension during system startup, your patch is placed in the system heap and applies to all applications. For example, if you patch the `FillRect` procedure with the patch `MyPatchFillRect` from a system extension, the `MyPatchFillRect` patch is executed every time the `FillRect` procedure is called, no matter which application calls it.

A-Line Instructions

When your application calls a Toolbox or an Operating System routine, an A-line instruction is sent to the microprocessor. Each A-line instruction contains information about the called system software routine. Figure 8-5 shows the layout of an A-line instruction.

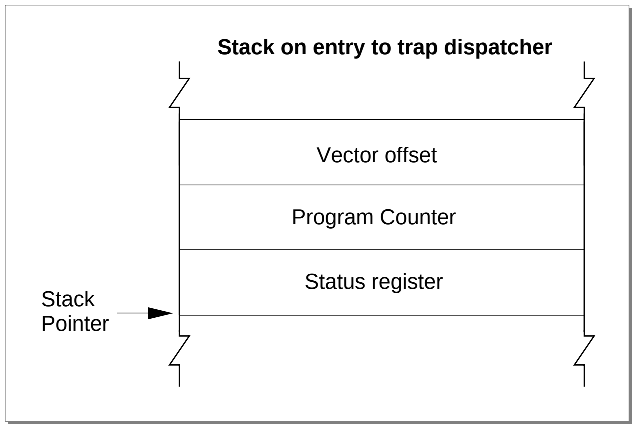
Figure 8-5 A-line instruction format



The high-order 4 bits of an A-line instruction have the hexadecimal value $A, hence the name A-line instruction. Bit 11 of the A-line instruction indicates the type of system software routine to be invoked: a value of 0 in bit 11 indicates an Operating System routine, a value of 1 in bit 11 indicates a Toolbox routine. The trap number in an A-line instruction is used as an index into the appropriate dispatch table. The meaning of the flags vary accordingly to the type of A-line instruction.

When your application calls a system software routine (thereby generating an exception), the microprocessor pushes an ***exception stack frame*** onto the stack. Figure 8-6 shows a typical exception stack frame. After pushing the exception stack frame on the stack, the microprocessor transfers control to the trap dispatcher.

Figure 8-6 Exception stack frame (on Macintosh computers with a MC68020 microprocessor or greater)



The trap dispatcher discards the status register and vector offset. Depending on whether the A-line instruction is used to invoke an Operating System routine or a Toolbox routine, the trap dispatcher deals with the stack and registers in two very different ways, as described in the next section, "A-line Instructions for Operating System Routines," and in the section "A-Line Instructions for Toolbox Routines" beginning on page 8-14.

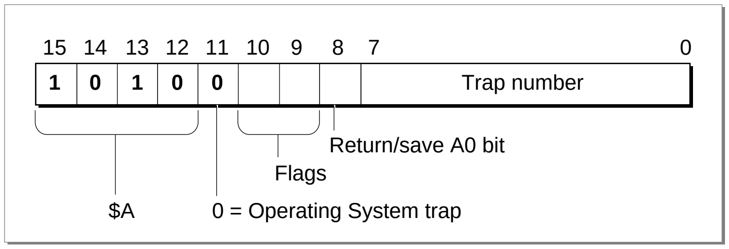
Note
The exception handler is located at address $28 on computers with an MC68000 microprocessor and at address $28 offset from the address in the microprocessor's Vector Base Register (VBR) on computers with other MC680x0 microprocessors. Consult the relevant microprocessor handbook for the precise details of exception handling on the MC680x0 microprocessor of interest to you. ◆

A-Line Instructions for Operating System Routines

An ***Operating System trap*** is an exception that is caused by an A-line instruction that executes an Operating System routine.

When dispatching an Operating System trap, the trap dispatcher extracts the trap number from the A-line instruction and uses it as an index into the Operating System trap dispatch table. The entry in the Operating System trap dispatch table contains the address of the desired Operating System routine. Figure 8-7 illustrates an A-line instruction for an Operating System routine.

Figure 8-7 An A-line instruction for an Operating System routine

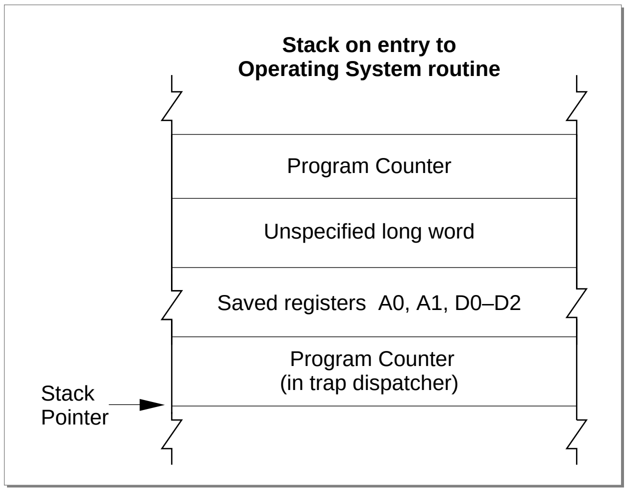


Bit 11 tells the trap dispatcher that this A-line instruction invokes an Operating System routine. Two flag bits, bit 10 and bit 9, are reserved for use by the Operating System routine itself and are discussed in detail in "Flag Bits" on page 8-14. Bit 8 indicates whether the value in register A0 is returned from the Operating System routine. If bit 8 is 0, the value in register A0 is returned from the Operating System routine. If bit 8 is 1, the value in register A0 is not returned by the Operating System routine. As previously described, the trap number is in bits 7–0 and is used to determine which of the 256 possible Operating System routines is executed.

For example, a call to the Operating System function `GetPtrSize` is translated to the A-line instruction $A021. This A-line instruction causes the microprocessor to transfer control to the trap dispatcher, which deals with any instruction of the form $A*xxx*. The trap dispatcher first saves registers D0, D1, D2, A1, and, if bit 8 is 0, A0. The trap dispatcher places the A-line instruction itself into the low-order word of register D1 so that the Operating System routine can inspect the flag bits. Next, the trap dispatcher examines the other bits in the A-line instruction. The value (0) of bit 11 indicates that `GetPtrSize` is an Operating System routine, and that the value in bits 7–0 is the index into the Operating System trap dispatch table. The trap dispatcher uses the index (which is 33 in this example) to find the address of the `GetPtrSize` function in the Operating System trap dispatch table. When the address is found, the trap dispatcher transfers control to the `GetPtrSize` function.

Figure 8-8 illustrates the stack after the trap dispatcher has transferred control to an Operating System routine.

Figure 8-8 The stack on entry to an Operating System routine


Stack on entry to Operating System routine
Program Counter
Unspecified long word
Saved registers A0, A1, D0–D2
Program Counter (in trap dispatcher)
Stack Pointer

The Operating System routine may alter any of the registers D0–D2 and A0–A2, but it must preserve registers D3–D7 and A3–A6. The Operating System routine may return information in register D0 (and A0 if bit 8 is set). To return to the trap dispatcher, the Operating System routine executes the RTS (return from subroutine) instruction.

When the trap dispatcher resumes control, first it restores the value of registers D1, D2, A1, A2, and, if bit 8 is 0, A0. The values in registers D0 and, if bit 8 is 1, in A0 are not restored.

Calling Conventions for Register-Based Routines

Register-based routines receive their parameters from microprocessor registers, and they pass their results in microprocessor registers. Virtually all Operating System routines are register-based routines.

An Operating System routine returns information only in registers D0 and, if bit 8 is 1, A0. The stack and all other registers are unchanged.

Many Operating System routines return a result code in the low-memory word of register D0 to report whether the requested operation was performed successfully. A result code of 0 indicates that the routine completed successfully; any other value typically indicates an error. Just before the trap dispatcher finishes execution, it tests the low-order word of register D0 with a TST.W instruction to set the condition codes of the microprocessor.

Note
Calling conventions for PowerPC microprocessor-based Macintosh computers are different from the calling conventions described for in this section. For information about calling conventions for PowerPC processor-based Macintosh computers, see *Inside Macintosh: PowerPC System Software*. ◆

Parameter-Passing Conventions for Operating System Routines

By convention, register-based routines normally use register A0 for passing addresses (such as pointers to data objects) and register D0 for other data values (such as integers).

For routines that take more than two parameters, the parameters are normally collected in a parameter block in memory and a pointer to the parameter block is passed in register A0. See the description of an individual routine in the appropriate *Inside Macintosh* book for exact details.

Function Results

Most Operating System functions return their function result (or result code) in register D0. Parameters are returned through register A0, usually as a pointer to a parameter block.

Whether the trap dispatcher preserves register A0 depends on the setting of bit 8 in the A-line instruction. If bit 8 is 0, the trap dispatcher saves and restores register A0; if it’s 1, the routine passes back register A0 unchanged. Thus, bit 8 of the A-line instruction should be set to 1 only for those routines that use register A0 to return information. The trap macros automatically set this bit correctly for each routine.

To see in which register the function passes the function result, see the description of the individual function in the appropriate *Inside Macintosh* book.

Flag Bits

Many Operating System routines use the flag bits in an A-line instruction to encode additional information used by the routine. For example, the A-line instructions that invoke Memory Manager routines define the two flag bits like this:

Bit	Explanation
9	If 1, initialize all bytes in the allocated memory to 0. If 0, do not initialize all bytes in the allocated memory to 0.
8	If 1, allocate memory from the system heap. If 0, allocate memory from the application heap.

These two bits are defined in assembly language as:

```
CLEAR    EQU      $200     ;initialize block to zero
SYS      EQU      $400     ;use the system heap
```

When used with a Memory Manager A-line instruction, these modifiers cause flag bits 9 and 10, respectively, to be set. They could be used in an assembly-language instruction sequence like

```
         MOVEQ    #124,D0        ;need 124 bytes
         _NewPtr     SYS,CLEAR   ;allocate requested memory in
                                 ; system heap and initialize to
                                 ; zeroes
```

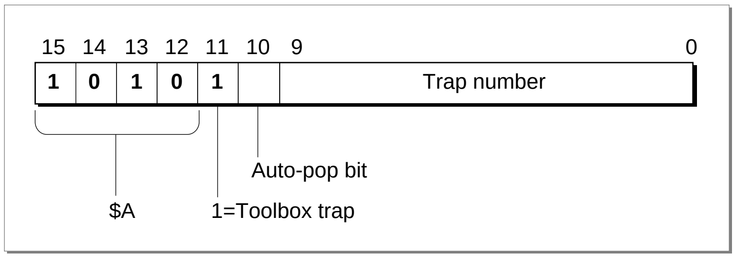
The `SYS` modifier specifies allocation from the system heap, regardless of the value of the global variable `TheZone`, and the `CLEAR` modifier specifies that the Memory Manager should initialize the block contents to zero. For further details, consult *Inside Macintosh: Memory*.

A-Line Instructions for Toolbox Routines

A ***Toolbox trap*** is an exception that is caused by an A-line instruction that executes a Toolbox routine.

When dispatching a Toolbox trap, the trap dispatcher extracts the trap number from the A-line instruction and uses it as an index into the Toolbox trap dispatch table. The index points to the entry in the Toolbox trap dispatch table that contains the address of the desired Toolbox routine. Figure 8-9 illustrates an A-line instruction that is used to access a Toolbox routine.

Figure 8-9 An A-line instruction for a Toolbox routine

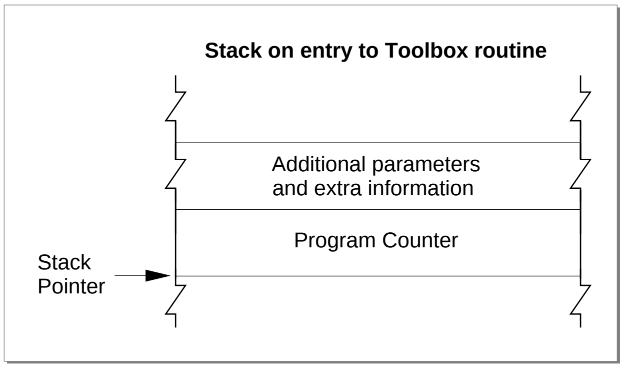


Bit 11 tells the trap dispatcher that this A-line instruction is used to access a Toolbox routine. Bit 10 is the auto-pop bit. Bits 9–0 contain the trap number which, as previously described, determine which of the 1024 possible Toolbox routines is executed. The auto-pop bit is described in detail in "The Auto-Pop Bit" on page 8-20.

For example, a call to the Toolbox function `WaitNextEvent` is translated to the A-line instruction $A860. This A-line instruction causes the microprocessor to transfer control to the trap dispatcher, which deals with any instruction of the form $A*xxx*. The trap dispatcher examines the other bits in the A-line instruction. The value (0) of bit 11 indicates that `WaitNextEvent` is a Toolbox routine and that the value in bits 9–0 is the index into the Toolbox trap dispatch table. The trap dispatcher uses the index (which is $60 in this example) to find the address of the `WaitNextEvent` function in the Toolbox trap dispatch table. When the address is found, the trap dispatcher transfers control to the `WaitNextEvent` function.

Figure 8-10 illustrates the stack after the trap dispatcher has transferred control to a Toolbox routine.

Figure 8-10 Stack when entering a Toolbox routine


Stack on entry to Toolbox routine
Additional parameters and extra information
Program Counter
Stack Pointer

The value of the Program Counter that is left on the stack before entry to the Toolbox routine points to the instruction that is executed after the completion of the Toolbox routine.

After the trap dispatcher completes execution, the internal status of the stack is restored, and normal execution resumes from the point at which processing was suspended.

A Toolbox routine changes the Stack Pointer in register A7 and pops the return address and any input parameters. A routine might also alter registers D0–D2, A0, and A1.

▲ **WARNING**
Some Toolbox routines (for example the `LongMul` procedure described in the chapter “Mathematical and Logical Utilities” in this book) preserve more than the required set of registers. However, you should assume all of registers D0–D2, A0, and A1 are altered by Toolbox routines. ▲

Calling Conventions for Stack-Based Routines

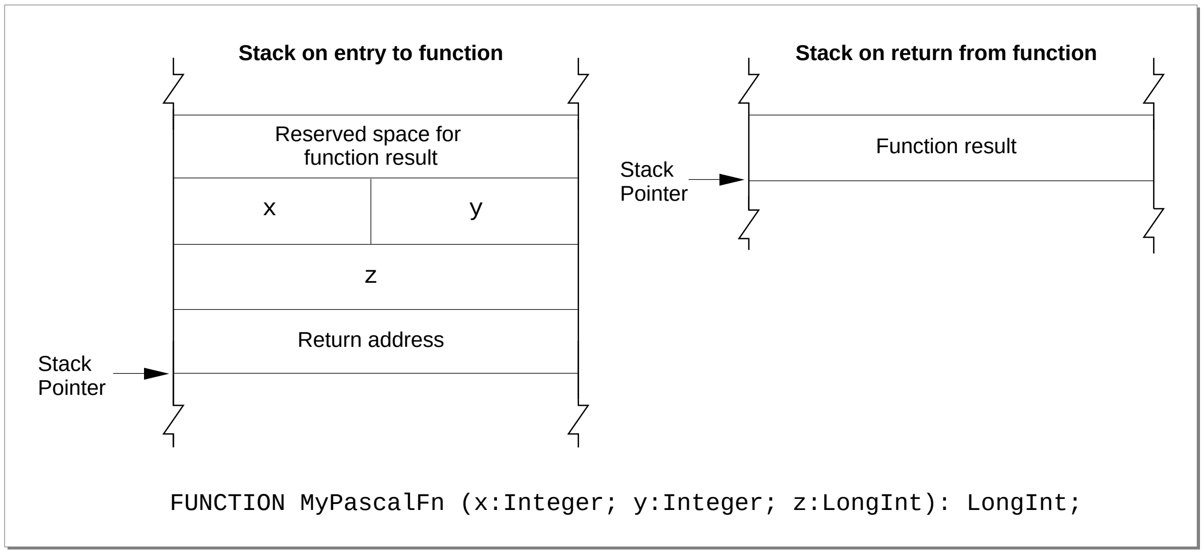
Stack-based routines receive their parameters on the stack and return their results on the stack. Virtually all Toolbox routines are stack-based routines.

Most Toolbox routines follow Pascal calling conventions; that is, Toolbox routine parameters are evaluated from left to right and are pushed onto the stack in the order in which they are evaluated. Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed on the stack. The caller is responsible for removing the result from the stack after the call.

Note
Calling conventions for PowerPC microprocessor-based Macintosh computers are different from the calling conventions described in this section. For information about calling conventions for PowerPC processor-based Macintosh computers, see *Inside Macintosh: PowerPC System Software*. ◆

Figure 8-11 illustrates Pascal calling conventions. In this example, a routine calls the application-defined function `MyPascalFn`. When the application calls the function `MyPascalFn`, the application must first make room on the stack for the function result, then push the parameters on the stack in left-to-right order.

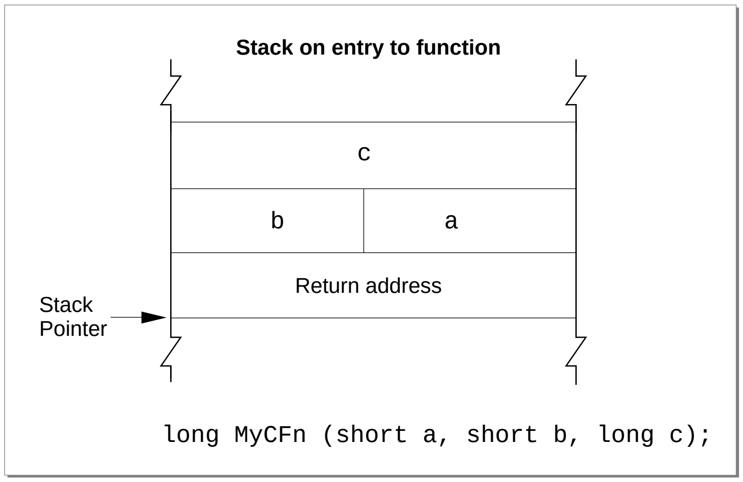
Figure 8-11 Pascal calling convention


Stack on entry to function
Reserved space for function result
x
y
z
Return address
Stack Pointer
Stack on return from function
Function result
Stack Pointer

```
FUNCTION MyPascalFn (x:Integer; y:Integer; z:LongInt): LongInt;
```

Figure 8-12 illustrates C calling conventions. In this example, a routine calls the application-defined function `MyCFn`. When the application calls the function `MyCFn`, the application pushes the parameters on the stack in right-to-left order. The function result is returned in register D0, and not on the stack.

Figure 8-12 C calling convention



```
long MyCFn (short a, short b, long c);
```

Parameter-Passing Conventions for Toolbox Routines

All variable parameters (parameters of type VAR) are passed as pointers to the actual storage location. In the case of byte-sized types, parameters of type VAR may have odd values.

Nonvariable parameters are passed in different ways, depending on the type of the parameter. Values of type `Boolean`, elements of an enumerated type with fewer than 128 elements, and subranges within the range –128 to 127 are passed as signed byte values. Values of type `Integer` and, `Char` and all other enumerations and subranges are passed as signed word values. Pointers and values of type `LongInt` are passed as signed 32-bit values. Table 8-1 summarizes the parameter-passing conventions.

Table 8-1 Toolbox parameter-passing conventions

Parameter type	Data object pushed on stack
Boolean	Byte: range 0 to 1
Char	16 bits: range 0 to 255
Integer	16 bits: range –32768 to 32767
LongInt	32 bits
Pointer	32 bits
Enumeration: range 0 to 127	Byte: range 0 to 127
Enumeration: range 0 to 32767	16 bits: range 0 to 32767
Subrange: range –128 to 127	16 bits: range –128 to 127
Subrange: range –32768 to 32767	Word: range –32768 to 32767
Real	Address of `Extended` copy
Double	Address of `Extended` copy
Comp	Address of `Extended` copy
Extended	Address of argument
ARRAY, RECORD, string ≤ 4 bytes	Value (word or long word)
ARRAY, RECORD, string > 4 bytes	Address of value
SET	SET value rounded to whole number of words

A parameter of type SET is passed by rounding its size up to the next whole word, if necessary, then pushing its value so that the lowest-order word is pushed last. In the case of a byte-size SET, the called procedure accesses only the low-order half of the word that is pushed.

Note
A byte pushed on the stack occupies the high-order byte of the word allocated for it, according to conventions for the MC680x0 microprocessors. ◆

▲ **WARNING**
A value of type `Char` is passed as a word value. The value occupies the low-order half of the word. ▲

Function Results

Function results are returned by value or by address on the stack. Space for the function result is allocated by the caller before the parameters are pushed. The caller is responsible for removing the result from the stack after the call.

For types `Boolean`, `Char`, and `Integer` and for enumerated and subrange types, the caller allocates a word on the stack to make space for the function result. Values of type `Boolean`, enumerated types with fewer than 128 elements, and subranges within the range –128 to 127 are returned as signed byte values. The value is placed in the high-order byte of the word.

Values of type `Integer` and `Char` and all enumerated and subrange types not covered above are returned as signed word values.

Pointers and values of type `LongInt` are returned as signed 32-bit values. Values of type `Real` are returned as 32-bit real values. For types whose values are greater than 4 bytes in size, the caller pushes a pointer to a temporary location into which the function places the result; these types include `Double` (8 bytes), `Comp` (8 bytes), and `Extended` (10 or 12 bytes); types `SET`, `ARRAY`, `RECORD`; and strings greater than 4 bytes in size.

For a 1-byte `SET`, for types `SET`, `ARRAY`, and `RECORD`, and for strings whose size is one word, the caller allocates a word on the stack. For types `SET`, `ARRAY`, and `RECORD` and strings whose size is two words, the caller allocates a long word on the stack.

The conventions for returning results of functions are summarized in Table 8-2.

Table 8-2 Conventions for returning results from Toolbox functions

Function result type	Data object left on stack or returned through pointer on stack
Boolean	Byte: range 0 to 1
Char	16 bits: range 0 to 255
Integer	16 bits: range –32768 to 32767
LongInt	32 bits
Pointer	32 bits
Enumeration: range 0 to 127	Byte: range 0 to 127
Enumeration: range 0 to 32767	16 bits: range 0 to 32767

continued

Table 8-2 Conventions for returning results from Toolbox functions (continued)

Function result type	Data object left on stack or returned through pointer on stack
Subrange: range –128 to 127	Byte: range –128 to 127
Subrange: range –32768 to 32767	16 bits: range –32768 to 32767
Real	Real
Double	`Double` at address given by pointer
Comp	`Comp` at address given by pointer
Extended	`Extended` at address given by pointer
`ARRAY`, `RECORD`, string ≤ 4 bytes	Value (word or long word)
`ARRAY`, `RECORD`, string > 4 bytes	Value at address given by pointer
`SET`: one byte	Byte value
`SET`: one word	16-bits value
`SET`: two words	32-bits value
`SET` > two words	Value at address given by pointer

Note
A 1 byte-size return value occupies the high-order byte of the word allocated for it. ◆

The Auto-Pop Bit

The ***auto-pop bit*** is bit 10 in an A-line instruction for a Toolbox routine. Some language systems prefer to generate jump-subroutine calls (`JSR`) to intermediate routines, called glue routines, which then call Toolbox routines instead of executing the Toolbox routine directly. This glue method would normally interfere with Toolbox traps because the return address of the glue subroutine is placed on the stack between the Toolbox routine's parameters and the address of the place where the glue routine was called from (where control returns once the Toolbox routine has completed execution).

The auto-pop bit forces the trap dispatcher to remove the top 4 bytes from the stack before dispatching to the Toolbox routine. After the Toolbox routine completes execution, control is transferred back to the place where the glue routine was called from, not back to the glue routine.

Most development environments, including MPW, do not use this feature.

About Trap Macros

A ***trap macro*** is an assembly-language macro that assembles into an A-line instruction, used for calling a Toolbox or Operating System routine from assembly language. The names of all trap macros begin with the underscore character (_), followed by the name of the corresponding routine. As a rule, the macro name is the same as the name used to call the routine from Pascal. For example, to call the Window Manager function `NewWindow`, you should use an instruction with the macro name `_NewWindow`. There are some exceptions, however, in which the spelling of the macro differs from the name of the Pascal routine itself; these are noted in the documentation for the individual routines.

Trap macros for Toolbox routines take no arguments; any parameters must be pushed on the stack before invoking the routine. See "Calling Conventions for Stack-Based Routines" on page 8-16 for more information. Trap macros for Operating System routines may have as many as three optional arguments. The first argument, if present, is used to load a register with a parameter value for the routine you're calling. The remaining arguments control the settings of the various flag bits in the A-line instruction.

About Routine Selectors

A routine selector is a value that is pushed on the stack to select a particular routine from a group of routines to be executed. Many trap macros take routine selectors. For example, the trap macro `_HFSDispatch` has the possibility of calling 42 different system software routines. Hence, the trap macro has 42 different routine selectors. The routine selector that is passed on the stack (for `_HFSDispacth` to access) selects which of the 42 software routines `_HFSDispatch` executes.

Most system software routines that are accessed through a trap macro and a routine selector also have a corresponding macro that expands to call the original trap macro and automatically puts the correct routine selector on the stack. For example, the trap macro `_GetCatInfo` expands to call `_HFSDispatch` and places the selector $0009 on the stack after the parameters.

Using the Trap Manager

You can use the Trap Manger to read from and write to a trap dispatch table. To read an address from a trap dispatch table, you can call the `NGetTrapAddress`, `GetOSTrapAddress`, or `GetToolboxTrapAddress` functions. To write an address to a trap dispatch table, you can use the `NGetTrapAddress`, `GetOSTrapAddress`, or `GetToolboxTrapAddress` procedures.

This section shows how you can use the Trap Manager to

- determine if a system software routine is available
- patch a system software routine

Determining If a System Software Routine is Available

You can use the Trap Manager to determine the availability of system software routines.

The Gestalt Manager, introduced in System 6.0.4 and discussed in the chapter "Gestalt Manager" in this book, is the primary tool for querying the system about its features. But if you expect your application to run on a system older than System 6.0.4, the Gestalt Manager may not be available.

The example in this section shows how you can use the Trap Manager to check whether a particular system software routine is available on the installed system.

At startup time, system software places the address of the `Unimplemented` procedure into all entries of each trap dispatch table that do not contain an address of a Toolbox or Operating System routine (or the address of a come-from patch). Listing 8-1 illustrates how you can use these `Unimplemented` addresses to determine whether a particular system software routine is available on the user's system. If a system software routine is available, its address differs from the address of the `Unimplemented` procedure.

Listing 8-1 Determining if a system software routine is available

```
FUNCTION MySWRoutineAvailable (trapWord: Integer): Boolean;
VAR
   trType:  TrapType;
BEGIN
   {first determine whether it is an Operating System or Toolbox routine}
   IF ORD(BAND(trapWord, $0800)) = 0 THEN
      trType := OSTrap
   ELSE
      trType := ToolTrap;
   {filter cases where older systems mask with $1FF rather than $3FF}
   IF (trType = ToolTrap) AND (ORD(BAND(trapWord, $03FF)) >= $200) AND
      (GetToolboxTrapAddress($A86E) = GetToolboxTrapAddress($AA6E)) THEN
      MySWRoutineAvailable := FALSE
   ELSE
      MySWRoutineAvailable := (NGetTrapAddress(trapWord, trType) <>
                              GetToolboxTrapAddress(_Unimplemented));
END;
```

Note
Macintosh Plus and Macintosh SE computers with system software prior to System 7 masked their trap numbers with $1FF in the `GetToolboxTrapAddress` function so that the address of A-line instruction $AA6E (whether implemented or not) would be the same as A-line instruction $A86E, which invokes the `InitGraf` routine. ◆

You can use the application-defined procedure `MySWRoutineAvailable` to check for system software routines not supported by the Gestalt Manager. A notable example is the `WaitNextEvent` function, which has never had `Gestalt` selectors. Listing 8-2 shows two common uses of the application-defined `MySWRoutineAvailable` procedure.

Listing 8-2 Determining whether `WaitNextEvent` and `Gestalt` are available

```
VAR
   gHasWNE, gHasGestalt:   Boolean;

   {check for the availability of WaitNextEvent}
   gHasWNE := MySWRoutineAvailable(_WaitNextEvent);
   {check for the availability of Getstalt}
   gHasGestalt := MySWRoutineAvailable(_Gestalt);
```

Patching a System Software Routine

Although this chapter describes patching in some depth, you should rarely, if ever, find a need to use patches in an application. The primary purposes of patches, as their name suggests, are to fix problems and augment routines in ROM code. The examples in this section are only included for the sake of completeness.

Listing 8-3 illustrates a patch for the `SysBeep` Operating System procedure. When `SysBeep` is called, this application-defined patch `MySysBeep` is executed before transferring control to the original `SysBeep` procedure.

Listing 8-3 Patching the `SysBeep` Operating System procedure

```
PROCEDURE MySysBeep (duration: Integer);
VAR
   oldPort:    GrafPtr;
   wMgrPort:   GrafPtr;
   i:          Integer;
BEGIN
   GetPort(oldPort);
   GetWMgrPort(wMgrPort);
   SetPort(wMgrPort);
   FOR := 3 DOWNTO 0 DO BEGIN
      InvertRect(wMgrPort^.portBits.bounds);
   END;
   SetPort(oldPort);
END; {of MySysBeep}
```

To transfer control to the next routine in the daisy chain (in this example the original `SysBeep` procedure), the application-defined `MyInstallAPatch` procedure (Listing 8-5) uses the application-defined procedure `MyFollowDaisyChain`, shown in Listing 8-4. The `MyFollowDaisyChain` duplicates the parameter for the `SysBeep` procedure and then pushes the address of the `SysBeep` procedure on the stack. Listing 8-4 shows the application-defined procedure `MyFollowDaisyChain`.

Listing 8-4 Jumping to the next routine in the daisy chain

```
MyFollowDaisyChain PROC EXPORT
IMPORT MYSYSBEEP
   BRA.S    @2
@1 DC.L     $50FFC001
@2 MOVE.W   $4(A7),-(A7)   ;duplicate the parameters
   MOVE.L   @1,-(A7)       ; and push the chain link
   BRA.S    MYSYSBEEP
   NOP
ENDPROC
END
```

The application-defined procedure `MyInstallAPatch` in Listing 8-5 installs a patch into the daisy chain (in this example, the `MySysBeep` patch). First, the procedure calls the `NGetTrapAddress` function to get the address of the next routine in the daisy chain. This address could be the address of another patch or the system software routine. Next, `MyInstallAPatch` calls the `NSetTrapAddress` procedure to put the address of the new patch (in this example, the address of `MySysBeep` patch) into the trap dispatch table.

Listing 8-5 Installing a patch

```
PROGRAM MyPatchInstaller;
USES  Memory, ToolIntf, OSIntf, OSUtils,Windows,
      ToolUtils, Traps, Resources, SamplePatch;
TYPE
PatchCodeHandle = ^PatchCodePtr;
PatchCodePtr = ^PatchCodeHeader;
PatchCodeHeader =
   RECORD
      branch:           Integer;
      oldTrapAddress:   LongInt;
   END;
PROCEDURE MyFollowDaisyChain (duration: Integer); EXTERNAL;
PROCEDURE MyInstallAPatch (trapNumber: Integer; tType: TrapType;
                           pPatchCode: PatchCodePtr);
BEGIN
   pPatchCode^.oldTrapAddress := NGetTrapAddress(trapNumber,
                                                 tType);
   NSetTrapAddress (ORD4(pPatchCode), trapNumber, tType);
END; {of MyInstallAPAtch}
```

```
BEGIN
   InitGraf (@qd.thePort);
   InitFonts;
   InitWindows;
   MyInstallAPatch(_SysBeep, ToolTrap,
                   PatchCodePtr(@MyFollowDaisyChain));
   SysBeep(1);
END. {of MyPatchInstaller}
```

Note
The `MyInstallAPatch` procedure used in this example was designed to install both Operating System and Toolbox patches; it uses the `NGetTrapAddress` and `NSetTrapAddress` routines. The `NGetTrapAddress` and `NSetTrapAddress` routines both need a parameter that indicates which type of routine is being patched, an Operating System or Toolbox routine. ◆

Trap Manager Reference

This section describes the routines provided by the Trap Manager. You can use these routines to

- access an address in a trap dispatch table
- install a patch address into a trap dispatch table

This section also documents the `Unimplemented` procedure.

Routines

This section describes the routines provided by the Trap Manager.

Accessing Addresses From the Trap Dispatch Tables

You can access the address of a system software routine by using the `GetOSTrapAddress`, `GetToolboxTrapAddress` or `NGetTrapAddress` function. The `GetOSTrapAddress` function retrieves only an Operating System routine address, and the `GetToolboxTrapAddress` retrieves only a Toolbox routine address. The `NGetTrapAddress` function is the most general of these functions; you can use the function to retrieve the address of an Operating System routine or a Toolbox routine.

GetOSTrapAddress

You can use the `GetOSTrapAddress` function to access the address of an Operating System routine, that is located in the Operating System trap dispatch table.

```
FUNCTION GetOSTrapAddress (trapNum: Integer): LongInt;
```

`trapNum`	Operating System A-line instruction or a trap number. If you specify an Operating System A-line instruction, the function extracts the trap number for you.

DESCRIPTION

The `GetOSTrapAddress` function returns the address of the Operating System routine specified by the `trapNum` parameter. If the desired Operating System routine is not supported on the installed system software, the `GetOSTrapAddress` function returns the address of the `Unimplemented` procedure. The `trapNum` parameter should contain a trap number in bits 0–7. `GetOSTrapAddress` masks the irrelevant high-order bits. A `GetOSTrapAddress(trapNum)` function call performs the same operation as a `NGetTrapAddress(trapNum, OSTrap)` function call.

SEE ALSO

For more information about the `Unimplemented` procedure, see page 8-29. For information about the `NGetTrapAddress` function, see page 8-27.

GetToolboxTrapAddress

You an use the `GetToolboxTrapAddress` function to access the address of a Toolbox routine, which is located in the Toolbox trap dispatch table. The `GetToolboxTrapAddress` function is also available as the `GetToolTrapAddress` function.

```
FUNCTION GetToolboxTrapAddress (trapNum: Integer): LongInt;
```

`trapNum`	Toolbox A-line instruction or a trap number. If you specify a Toolbox A-line instruction, the function extracts the trap number for you.

DESCRIPTION

The `GetToolboxTrapAddress` function returns the address of the Toolbox routine specified by the `trapNum` parameter. If the desired Toolbox routine is not supported on the installed system software, the `GetToolboxTrapAddress` function returns the address of the `Unimplemented` procedure. The `trapNum` parameter should contain a trap number in bits 0–9. `GetToolboxTrapAddress` masks the irrelevant high-order bits. A `GetToolboxTrapAddress(trapNum)` function call performs the same operation as a `NGetTrapAddress(trapNum, ToolTrap)` function call.

SEE ALSO

For more information about the `Unimplemented` procedure, see page 8-29. The `NGetTrapAddress` function is described next. For an example of how to use the `GetToolboxTrapAddress` function, see Listing 8-1 on page 8-22.

NGetTrapAddress

You can use the `NGetTrapAddress` function to retrieve the address of either an Operating System routine or a Toolbox routine.

```
FUNCTION NGetTrapAddress (trapNum: Integer; tTyp: TrapType)
                                  :LongInt;
```

`trapNum`	A-line instruction or a trap number. If you specify an A-line instruction, the function extracts the trap number for you.
`tTyp`	The trap type. If you supply the `tTyp` parameter with the constant `OSTrap`, the `NGetTrapAddress` function retrieves the address from the Operating System trap dispatch table. If you supply `tTyp` parameter with the constant `ToolTrap`, the `NGetTrapAddress` function retrieves the address from the Toolbox trap dispatch table.

DESCRIPTION

The `NGetTrapAddress` function returns the address of the system software routine specified by the `tTyp` and `trapNum` parameters. If `tTyp` is `OSTrap`, the `NGetTrapAddress` function retrieves the address from the Operating System trap dispatch table. If `tTyp` is `ToolTrap`, the `NGetTrapAddress` function retrieves the address from the Toolbox trap dispatch table. If the desired system software routine is not supported on the installed system software, `NGetTrapAddress` returns the address of the `Unimplemented` procedure. The `trapNum` parameter should contain a trap number in bits 0–7 if `tTyp` is `OSTrap`, and in bits 0–9 if `tTyp` is `ToolTrap`.
The `trapNum` parameter may have any word value; its irrelevant high-order bits are masked according to the value of the `tTyp` parameter.

Note
If the system software routine has a come-from patch, the `NGetTrapAddress` function returns the address of the routine immediately following the come-from patch. ◆

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for the `_GetTrapAddress` macro are

Registers on entry

D0	An A-line trap word

Registers on exit

A0	Address of next routine in the daisy chain (a system software routine or a patch)

When calling the `_GetTrapAddress` macro, you set bit 9 of the A-line instruction to indicate a "new" system; that is, any version since the Macintosh Plus or Macintosh 512K. You use bit 10 to indicate whether the trap in question is a Toolbox routine (by setting bit 10 to 1) or an Operating System routine (by setting bit 10 to 0). Macintosh development environments provide the modifier words `newTool` and `newOS` to be used as arguments in the `_GetTrapAddress` macro.

To obtain the address of a Toolbox trap whose number is in register D0, you use the macro

```
_GetTrapAddress newTool
```

This is equivalent to calling `NGetTrapAddress` (`trapNum, newTool`). The `trapNum` parameter is the A-line trap word placed in register D0 for the assembly-language call. Similarly, to obtain the address of an Operating System routine whose A-line trap word is in register D0, you use the macro

```
_GetTrapAddress newOS
```

This is equivalent to calling `NGetTrapAddress(trapNum, newOS)`.

SEE ALSO

For information about the `Unimplemented` procedure, see page 8-29. For information about the `NSetTrapAddress` function, see page 8-30.

Installing Patch Addresses Into the Trap Dispatch Tables

You can install the address of a patch into a trap dispatch table by using the `SetOSTrapAddress`, `SetToolboxTrapAddress`, or `NSetTrapAddress` procedure. The `SetOSTrapAddress` procedure installs a patch address into the Operating System trap dispatch table, and the `SetToolboxTrapAddress` installs a patch address into the Toolbox trap dispatch table. The `NSetTrapAddress` procedure is the most general of these procedures. You can use the `NSetTrapAddress` procedure to install a patch address into the Operating System trap dispatch table or into the Toolbox trap dispatch table.

SetOSTrapAddress

You can use the `SetOSTrapAddress` procedure to install an Operating System patch address into an Operating System trap dispatch table.

```
PROCEDURE SetOSTrapAddress (trapAddr: LongInt; trapNum: Integer);
```

`trapAddr`	The Operating System patch address.
`trapNum`	Operating System A-line instruction or a trap number. If you specify an Operating System A-line instruction, the function extracts the trap number (located in bits 0–7) for you.

DESCRIPTION

The `SetOSTrapAddress` procedure places the Operating System patch address specified by the `trapAddr` parameter into the Operating System trap dispatch table. The `trapNum` parameter specifies the location of the Operating System patch address in the Operating System trap dispatch table. The procedure call `SetOSTrapAddress(trapAddr, trapNum)` performs the same operation as a `NSetTrapAddress(trapAddr, trapNum, OSTrap)` procedure call.

Note
If the system software routine that is being patched has any come-from patches, the `SetOSTrapAddress` procedure installs the address of the patch into the exit JMP instruction of the last come-from patch in the chain rather than into the trap dispatch table. ◆

SEE ALSO

For information about the `Unimplemented` procedure, see page 8-29. For more information about the `NSetTrapAddress` function, see page 8-30.

SetToolboxTrapAddress

You can use the `SetToolboxTrapAddress` procedure to install a Toolbox patch address into the Toolbox trap dispatch table. The `SetToolboxTrapAddress` procedure is also available as the `SetToolTrapAddress` procedure.

```
PROCEDURE SetToolboxTrapAddress (trapAddr: LongInt;
                                 trapNum: Integer);
```

`trapAddr`	The Toolbox patch address.
`trapNum`	Toolbox A-line instruction or a trap number. If you specify a Toolbox A-line instruction, the function extracts the trap number (located in bits 0–9) for you.

DESCRIPTION

The `SetToolboxTrapAddress` procedure places the Toolbox patch address specified by the `trapAddr` parameter into the Toolbox trap dispatch table. The `trapNum` parameter specifies the location of the Toolbox patch address in the Toolbox trap dispatch table. The `SetToolboxTrapAddress(trapAddr, trapNum)` procedure performs the same operation as a `NSetTrapAddress(trapAddr, trapNum,` ToolTrap) procedure call.

Note
If the system software routine that is being patched has any come-from patches, the `SetToolboxTrapAddress` procedure installs the address of the patch into the exit JMP instruction of the last come-from patch in the chain rather than into the trap dispatch table. ◆

SEE ALSO

For information about the `Unimplemented` procedure, see page 8-29. The `NSetTrapAddress` function is described next.

NSetTrapAddress

You can use the `NSetTrapAddress` procedure to install a patch address into either an Operating System trap dispatch table or a Toolbox trap dispatch table.

```
PROCEDURE NSetTrapAddress (trapAddr: LongInt; trapNum: Integer;
                           tTyp: TrapType);
```

`trapAddr`	The patch address.
`trapNum`	A-line instruction or a trap number. If you specify a A-line instruction, the function extracts the trap number you.
`tTyp`	The trap type. If you supply the `tTyp` parameter with the constant `OSTrap`, the `NSetTrapAddress` procedure installs the address into the Operating System trap dispatch table. If you supply the `tTyp` parameter with the constant `ToolTrap`, the `NGetTrapAddress` function installs the address into the Toolbox trap dispatch table.

DESCRIPTION

The `NSetTrapAddress` procedure places the patch address specified by the `trapAddr` parameter into a trap dispatch table. Use the `tTyp` parameter to specify whether the patch address belongs in the Operating System trap dispatch table or the Toolbox trap dispatch table. If `tTyp` is `OSTrap`, the `NSetTrapAddress` procedure installs the address into the Operating System trap dispatch table. If `tTyp` is `ToolTrap`, the `NGetTrapAddress` function installs the address into the Toolbox trap dispatch table. Use the `trapNum` parameter to specify the location of the patch address in the dispatch table. The trap number may be any word value; its irrelevant high-order bits are masked according to the value of the `tTyp` parameter.

Note
If the system software routine that is being patched has a come-from patch, the `NSetTrapAddress` procedure installs the address of the patch into the exit JMP instruction of the come-from patch (rather than into the trap dispatch table). ◆

▲ **WARNING**
If the first 4 bytes of the `trapAddr` parameter is $60064EF9 (indicating a come-from patch), `NSetTrapAddress` triggers a system error. ▲

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry for the `_SetTrapAddress` macro are

Registers on entry

D0	An A-line trap word
A0	Address of next routine in the daisy chain (a system software routine or a patch)

When calling the `_SetTrapAddress` macro, you set bit 9 of the A-line trap word to indicate a “new” system; that is, any version since the Macintosh Plus or Macintosh 512K. You use bit 10 to indicate whether the system software routine that is being patched is a Toolbox routine (by setting bit 10 to 1) or an Operating System routine (by setting bit 10 to 0).

Macintosh development environments provide the modifier words `newTool` and `newOS` to be used as arguments in the `_SetTrapAddress` macro.

Given an A-line instruction in register D0 and a system software address in register A0, you set the Toolbox routine with the trap number in register D0 to have the address in A0, you use the macro

```
_SetTrapAddress newTool
```

This is equivalent to calling `NSetTrapAddress(trapAddr, trapNum, newTool)`. The `trapAddr` parameter is the address placed in register A0. The `trapNum` parameter is the A-line instruction placed in D0 for the assembly-language call. Similarly, to set the address of an Operating System trap whose A-line instruction is in register D0 to the address in register A0 you use the macro

```
_SetTrapAddress newOS
```

This is equivalent to calling `NSetTrapAddress(trapAddr, trapNum, newOS)`.

SEE ALSO

The `Unimplemented` procedure is described next. For information about the `NGetTrapAddress` function, see page 8-27. For an example of how to use the `NSetTrapAddress` function, see Listing 8-5 on page 8-24.

Detecting Unimplemented System Software Routines

This section describes the `Unimplemented` procedure. The address of this procedure is placed in all undefined entries of a trap dispatch table. When invoked, the `Unimplemented` procedure triggers a system error.

Unimplemented

The `Unimplemented` procedure triggers a system error when called.

```
PROCEDURE Unimplemented;
```

DESCRIPTION

The address of the `Unimplemented` procedure is at system startup time placed into all entries of each trap dispatch table that do not contain an address of a system software routine. When called, the `Unimplemented` procedure triggers the system error 12, `dsCoreErr`, which crashes the currently running application.

▲ **WARNING**
Your application should never use this procedure. ▲

Manipulating *One* Trap Dispatch Table (Obsolete Routines)

This section describes two obsolete Trap Manager routines: `GetTrapAddress` and `SetTrapAddress`. Though a description of the routines are included here, any use of these routines is discouraged.

GetTrapAddress

The `GetTrapAddress` function is obsolete and is documented here only for the sake of completeness.

```
FUNCTION GetTrapAddress (trapNum: Integer): LongInt;
```

`trapNum`	Toolbox A-line instruction or a trap number. If you specify an A-line instruction, the function extracts the trap number for you.

DESCRIPTION

The `GetTrapAddress` function was used when both the Operating System trap addresses and Toolbox trap addresses were located in the same trap dispatch table. Today, any system software routine with the trap number `$00` to `$4F`, `$54`, or `$57` is drawn from the Operating System dispatch table; any other software routine is taken from the Toolbox dispatch table.

▲ **WARNING**
The `GetTrapAddress` function is not supported under Power PC. ▲

▲ **WARNING**
The `GetTrapAddress` procedure ignores the high-order bits in the `trapNum` parameter; the procedure is not able to differentiate between Operating System routines and Toolbox routines. The `GetTrapAddress` procedure is not reliable on any computer today. ▲

SetTrapAddress

The `SetTrapAddress` procedure is obsolete, and is documented here only for the sake of completeness.

```
PROCEDURE SetTrapAddress (trapAddr: LongInt; trapNum: Integer);
```

`trapAddr`	The address of the system software routine.
`trapNum`	A-line instruction or a trap number. If you specify an A-line instruction, the function extracts the trap number you.

DESCRIPTION

The `SetTrapAddress` procedure was used when both the Operating System routine addresses and Toolbox routine adddresses were located in the same trap dispatch table. Today, any routine address with the trap number `$00` to `$4F`, `$54`, or `$57` is installed into the Operating System dispatch table; any other system software routine is installed into the Toolbox dispatch table.

▲ **WARNING**
The `SetTrapAddress` procedure is not supported under Power PC. ▲

▲ **WARNING**
The `SetTrapAddress` procedure ignores the high-order bits in the `trapNum` parameter; the procedure is not able to differentiate between Operating System routines and Toolbox routines. The `SetTrapAddress` procedure is not reliable on any computer today. ▲

Summary of the Trap Manager

Pascal Summary

Constants

```
CONST
   {Gestalt selectors}
   gestaltOSTable          = 'ostt';   {base of Operating System dispatch }
                                       { table}
   gestaltToolboxTable     = 'tbtt';   {base of Toolbox dispatch table}
   gestaltExtToolboxTable  = 'xttt';   {0, unless Toolbox dispatch table }
                                       { is disjoint, in which case base }
                                       { of upper half}

   {system errors triggered by the Trap Manager}
   dsCoreErr               = 12;       {unimplemented trap error}
   dsBadPatchHeader        = 83;       {attempt to install a come-from patch}
```

Data Types

```
TYPE TrapType     = (OSTrap, ToolTrap);
```

Routines

Accessing Addresses From the Trap Dispatch Tables

```
FUNCTION GetOSTrapAddress   (trapNum: Integer): LongInt;
{GetToolboxTrapAddress is also spelled as GetToolTrapAddress}
FUNCTION GetToolboxTrapAddress
                            (trapNum: Integer): LongInt;
FUNCTION NGetTrapAddress    (trapNum: Integer; tTyp: TrapType): LongInt;
```

Installing Patch Addresses Into the Trap Dispatch Tables

```
PROCEDURE SetOSTrapAddress  (trapAddr: LongInt; trapNum: Integer);
{SetToolboxTrapAddress is also spelled as SetToolTrapAddress}
```

```
PROCEDURE SetToolboxTrapAddress
                             (trapAddr: LongInt; trapNum: Integer);
PROCEDURE NSetTrapAddress    (trapAddr: LongInt; trapNum: Integer;
                              tTyp: TrapType);
```

Detecting Unimplemented System Software Routines

```
PROCEDURE Unimplemented;
```

Manipulating One Trap Dispatch Table (Obsolete Routines)

```
FUNCTION GetTrapAddress      (trapNum: Integer): LongInt;
PROCEDURE SetTrapAddress     (trapAddr: LongInt; trapNum: Integer);
```

C Summary

Constants

```
/*Gestalt selectors*/
#define gestaltOSTable        'ostt'   /*base of Operating System dispatch */
                                       /* table*/
#define gestaltToolboxTable   'tbtt'   /*base of Toolbox dispatch table*/
#define gestaltExtToolboxTable'xttt'   /*0, unless Toolbox dispatch table */
                                       /* is disjoint, in which case base */
                                       /* of upper half*/

/*values of TrapType*/
enum {OSTrap, ToolTrap};

/*system errors triggered by Trap Manager*/
enum {
   dsCoreErr         = 12,             /*unimplemented trap error*/
   dsBadPatchHeader  = 83              /*attempt to install come-from patch*/
};
```

Data Types

```
typedef unsigned char TrapType;
```

Routines

Accessing Addresses From the Trap Dispatch Tables

```
pascal long NGetTrapAddress
                            (short trapNum, TrapType tTyp);
pascal long GetOSTrapAddress
                            (short trapNum);
/*GetToolboxTrapAddress is also spelled as GetToolTrapAddress*/
pascal long GetToolboxTrapAddress
                            (short trapNum);
```

Installing Patch Addresses Into the Trap Dispatch Tables

```
pascal void NSetTrapAddress
                            (long trapAddr, short trapNum,
                             TrapType tTyp);
pascal void SetOSTrapAddress
                            (long trapAddr, short trapNum);
/*SetToolboxTrapAddress is also spelled as SetToolTrapAddress*/
pascal void SetToolboxTrapAddress
                            (long trapAddr, short trapNum);
pascal void SetToolTrapAddress
                            (long trapAddr, short trapNum);
```

Detecting Unimplemented System Software Routines

```
pascal void Unimplemented   (void);
```

Manipulating One Trap Dispatch Table (Obsolete Routines)

```
pascal long GetTrapAddress  (short trapNum);
pascal void SetTrapAddress  (long trapAddr, short trapNum);
```

Assembly-Language Summary

Constants

```
newOS          EQU   $200     ;access Operating System trap dispatch table;
newTool        EQU   $600     ;access Toolbox trap dispatch table
```

Trap Macros

Trap Macros Requiring Register Setup

Trap macro name	Registers on entry	Registers on exit
`_GetTrapAddress`	D0: trap number	A0: address of patch
`_SetTrapAddress`	D0: trap number A0: address of patch	
`_Unimplemented`		