# Procedure Calls 4

# CHAPTER 4
# PROCEDURE CALLS

This chapter describes the 80960SA/SB processor's procedure call and stack mechanism. It also describes the supervisor call mechanism, which provides a means of calling privileged procedures such as kernel services.

## TYPES OF PROCEDURE CALLS

The processor supports three types of procedure calls:

- Local call
- System call
- Branch and link

A local call uses the processor's call/return mechanism, in which a new set of local registers and a new frame on the stack are allocated for the called procedure. A system call is similar to a local call, except that it provides access to procedures through a system-procedure table. The most important use of a system call is to call privileged procedures, called *supervisor procedures*. A system call to a supervisor procedure is called a *supervisor call*. A branch and link is merely a branch to a new instruction with the return IP stored in a global register.

In this chapter, the call/return mechanism is introduced first and is followed by a discussion of how this mechanism is used to make local calls and system calls.

**NOTE**

The processor's interrupt- and fault-handling mechanisms use implicit procedure calls. Implicit calls to interrupt-handler and fault-handler procedures are described in detail in Chapters 5 and 6, respectively.

## CALL/RETURN MECHANISM

The processor's call/return mechanism has been designed to simplify procedure calls and to provide a flexible method for storing and handling variables that are local to a procedure.

Two structures support this mechanism: the local registers (on the processor chip) and the procedure stack (in memory). Figure 4-1 shows the relationship of the local registers to the procedure stack.

For each procedure, the processor automatically allocates a set of local registers and a frame on the procedure stack. Since the local registers are on-chip, they provide fast-access storage for local variables. If additional space for local variables is required, it can be allocated in the stack frame.

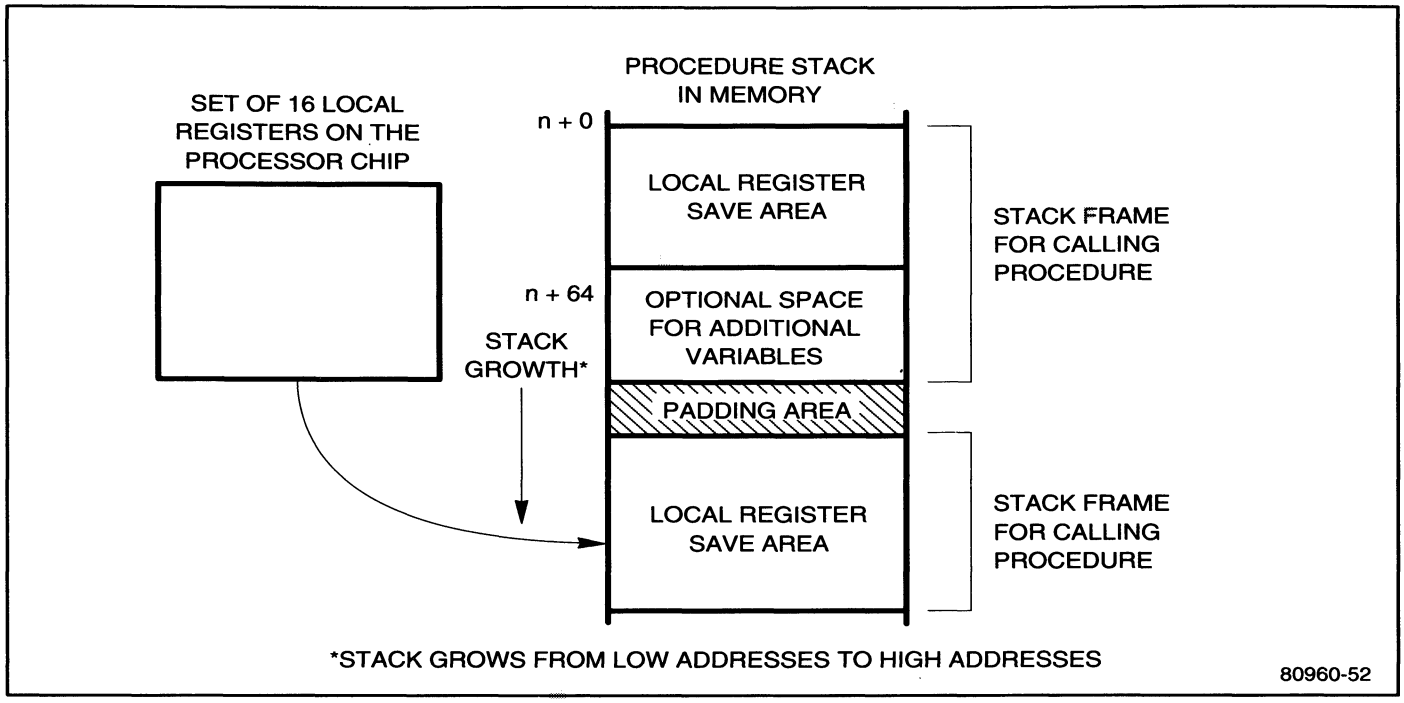


**Figure 4-1: Local Registers and Procedure Stack**

When a procedure call is made, the processor automatically saves the contents of the local registers and the stack frame for the calling procedure and sets up a new set of local registers and a new stack frame for the called procedure.

This procedure-call mechanism provides two benefits. First, it provides a structure for storing a virtually unlimited number of local variables for each procedure: the on-chip local registers provide quick access to often-used variables and the stack provides space for additional variables.

Second, a program does not have to explicitly save and restore the variables stored in the local registers and stack frames. The processor does this implicitly on procedure calls and on returns.

A detailed description of the call/return mechanism is given in the following paragraphs.

## Local Registers and the Procedure Stack

For each procedure, the processor allocates a set of 16 local registers. Three of these registers (r0, r1, and r2) are reserved for linkage information to tie procedures together. The remaining 13 local registers are available for general storage of variables.

The processor maintains a procedure stack in memory for use when performing local calls. This stack can be located anywhere in the address space and grows from low addresses to high addresses.

The stack consists of contiguous frames, one frame for each active procedure. As shown in Figure 4-2, each stack frame provides a save area for the local registers and an optional area for additional variables.

To increase the speed of procedure calls, the 80960SA/SB processor provides four sets of local registers. Thus, when a procedure call is made, the contents of the current set of local registers often do not have to be stored in the procedure stack. Instead, a new set of local registers is assigned to the called procedure. When the number of nested procedure calls exceeds the number of register sets, the processor automatically stores the contents of the oldest set of local registers on the stack to free up a set of local registers for the most recently called procedure.

Refer to the section later in this chapter titled "Mapping the Local Registers to the Procedure Stack" for further discussion of the relationship between the local-register sets and the procedure stack.

## Procedure Linking Information

Global register g15 (FP) and local registers r0 (PFP), r1 (SP), and r2 (RIP) contain information to link procedures together and to link the local registers to the procedure stack. The following paragraphs describe this linkage information.

## Frame Pointer

The FP is the address of the first byte of the current (topmost) stack frame. It is stored in global register g15. The 80960SA/SB processor aligns each new stack frame on a 64-byte boundary. Since the resulting FP always points to a 64-byte boundary, the processor ignores the 6 low-order bits of the FP and interprets them to be zero.

**NOTE**

The alignment boundary for new frames is defined by means of an implementation-dependent parameter called SALIGN. The relationship of SALIGN to the frame alignment boundary is described in Appendix F.

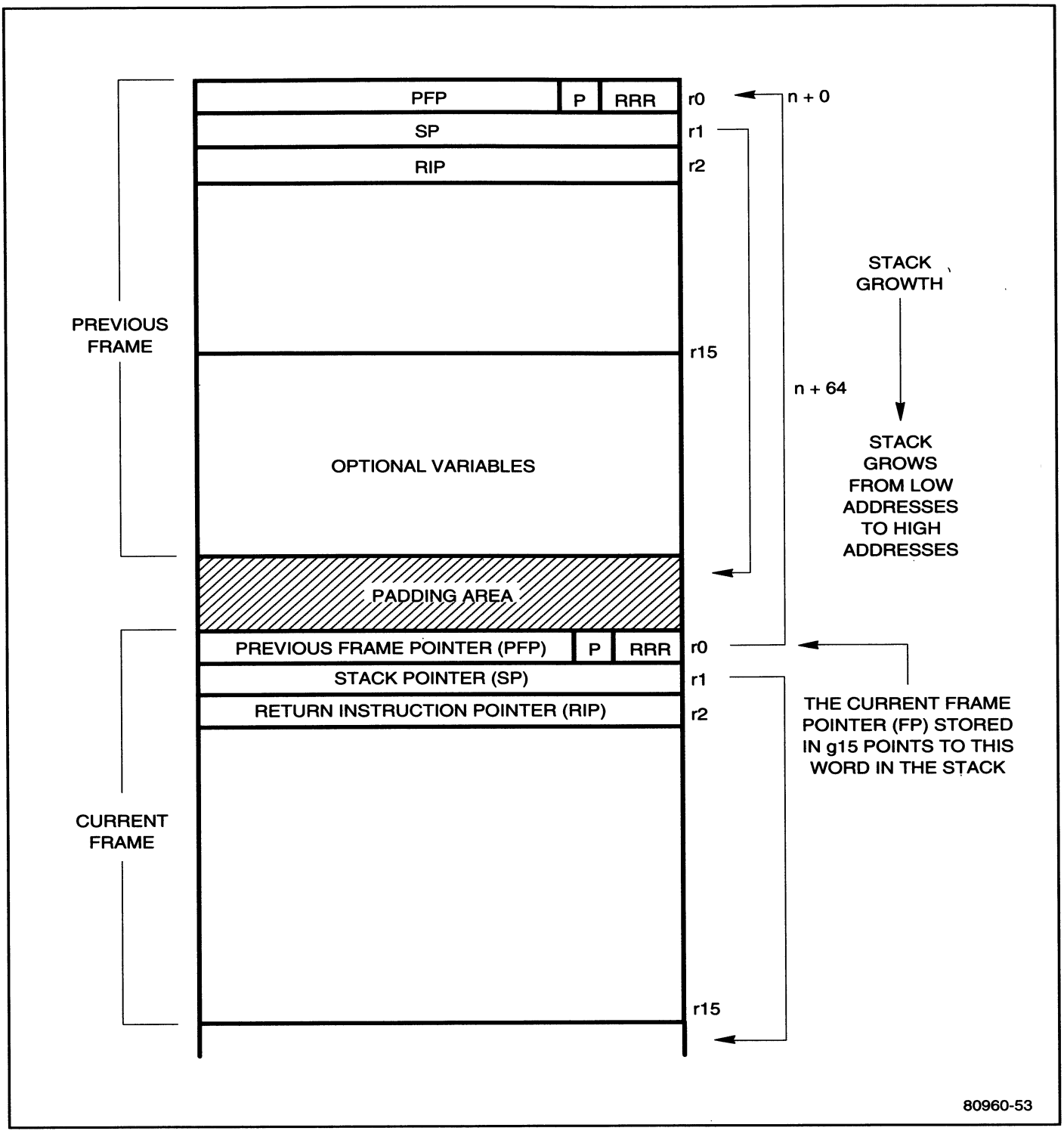


**Figure 4-2: Procedure Stack Structure**

### Stack Pointer

The SP is the address of the next available byte of the stack frame, which can also be thought of as the last byte of the stack frame plus one. It is stored in local register r1. The procedure stack grows upward (i.e., toward higher addresses). To determine the initial SP value, the processor adds 64 to the FP.

If additional space is needed on the stack for local variables, the SP may be incremented in one-byte increments. For example, the following instruction adds six words of additional space to the stack:

```
addo sp, 24, sp     #  sp ← sp + 24
```

With the Intel 80960SA/SB Assembler, the keyword "sp" stands for register r1.

When the processor creates a new frame on a procedure call, it will, if necessary, add a padding area to the stack so that the new frame starts on a 64 byte boundary. To create the padding area, the processor rounds off the SP for the current stack frame (the value in r1) to the next highest 64 byte boundary. This value becomes the FP for the new stack frame.

### Previous-Frame Pointer

The PFP is the address of the first byte of the previous stack frame. It is stored in local register r0. Since the 80960SA/SB ignores the 6 low-order bits of the FP, only the 26 most-significant bits of the PFP are stored here. The 4 least-significant bits of r0 are then used to store return status information.

### Return Status and Prereturn-Trace Information

Bits 0 through 2 of local register r0 contain return status information for the calling procedure and bit 3 contains the prereturn-trace flag. When a procedure call is made (either explicit or implicit), the processor records the call type in the return status field. The processor then uses this information to select the proper return mechanism when returning to the calling procedure.

Table 4-1 shows the encoding of the return status field according to the different types of calls that the processor supports. Of the four types of calls allowed, the fault call (described in Chapter 6) and the interrupt call (described in Chapter 5) are implicit calls that the processor initiates. The local call (described in this section) is an explicit call that a program initiates using the **call** or **callx** instruction. The supervisor call (described at the end of this chapter in the section titled "User-Supervisor Protection Model") is an explicit call that a program makes using the **calls** instruction.

The third column of Table 4-1 shows the type of a return action that the processor takes depending on the state of the return status field.

The processor records two versions of the supervisor call: one for when the trace-enable flag in the process controls is set prior to a supervisor call and one for when the flag is clear prior to the call. The trace controls are described in detail in Chapter 7.

The prereturn-trace flag is used in conjunction with the call-trace and prereturn-trace modes. If the call-trace mode is enabled when a call is made, the processor sets the prereturn-trace flag; otherwise it clears the flag. Then, if this flag is set and the prereturn-trace mode is enabled, a prereturn trace event is generated on a return before any actions associated with the return operation are performed. Refer to Chapter 7 for a detailed discussion of the interaction of the call-trace and prereturn-trace modes and the prereturn-trace flag.

**Table 4-1: Encoding of Return Status Field**

| Encoding | Call Type | Return Action |
|---|---|---|
| 000 | Local call or supervisor call made from the supervisor mode | Local return |
| 001 | Fault call | Fault return |
| 010 | Supervisor call from user mode, trace was disabled before call | Supervisor return, with the trace enable flag in the process controls set to 1 and the execution mode flag set to 0 |
| 011 | Supervisor call from user mode, trace was enabled before call | Supervisor return, with the trace enable flag in the process controls set to 0 and the execution mode flag set to 0 |
| 100 | reserved | |
| 101 | reserved | |
| 110 | reserved | |
| 111 | Interrupt call | Interrupt return |

## Return-Instruction Pointer

The RIP is the address of the instruction that the processor is to execute after returning from a procedure call. It is stored in local register r2. When the processor executes a procedure call it sets the RIP to the address of the instruction immediately following the procedure call instruction. (Refer to the section later in this chapter titled "Local Call Operation" for further information on the RIP.)

Since the processor uses the same procedure call mechanism to make implicit procedure calls to service faults and interrupts, programs should not use register r2 for purposes other than to hold the RIP.

## Mapping the Local Registers to the Procedure Stack

The availability of multiple register sets cached on the processor chip and the saving and restoring of these register sets in stack frames should be transparent to most programs. However, the following additional information about how the local registers and procedure stack are mapped to one another can help avoid problems.

Since the local-register sets reside on the processor chip, the processor will often not have to access the stack frame in the procedure stack, even though space has been allocated on the stack for the current frame. The processor only accesses the current frame in the procedure stack in the following instances:

1. to read or write variables other than those held in the local registers, or
2. to read local registers that were stored in the procedure stack when the number of nested procedures calls exceeded the number of local registers.

This method of mapping the local registers to the register-save areas in the procedure stack has several implications. First, storing information in a local register does not guarantee that it will be stored in its associated word in the current stack frame. Likewise, storing information in the first 16 words of a stack frame does not guarantee that the local registers associated with the stack frame are modified.

Second, if you try to read the contents of the current set of local registers through a memory access to the first 16 words of the current stack frame, you may not get the expected result. This is also true if you try to read the contents of a previously stored set of local registers through a memory address to its associated stack frame.

The processor automatically stores the contents of a local-register set into the register-save area of its associated stack frame only if procedure calls (local or supervisor) are nested deeper than the number of local-register sets.

Occasionally, it is necessary to have the contents of all local-register sets match the contents of the register-save areas in their associated stack frames. For example, when debugging software it may be necessary to trace the call history back through the nested procedures. This can not be done unless the cached local-register frames are flushed (i.e., written out to the procedure stack).

The processor provides the **flushreg** (flush local registers) instruction to allow voluntary flushing of the local registers. This instruction causes the contents of all the local-register sets, except the current set, to be written to their associated stack frames in memory.

Third, if you need to modify the PFP in register r0, you should precede this operation with the **flushreg** instruction, or else the behavior of the **ret** (return) instruction is not predictable.

Fourth, local registers should not be used for passing parameters between procedures. (Parameter passing is discussed in the following section.)

Fifth, when a set of local registers is assigned to a new procedure, the processor may not clear or initialize these registers. The initial contents of these registers are therefore unpredictable. Also, the processor does not initialize the local register-save area in the newly created stack frame for the procedure, so its contents are equally unpredictable.

## LOCAL CALL

A local call is made using either of two local call instructions: **call** and **callx**. These instructions initiate a procedure call using the call/return mechanism described earlier in this chapter.

The **call** instruction specifies the address of the called procedures as the IP plus a signed, 24-bit displacement (i.e., $-2^{23}$ to $2^{23} - 4$).

The **callx** instruction allows any of the addressing modes to be used to specify the procedure address. The *IP with displacement* addressing mode allows full 32-bit IP relative addressing.

The **ret** instruction initiates a procedure switch back to the last procedure that issued a call.

### Local Call Operation

During a local call, the processor performs the following operations:

1. Stores the RIP in current local register r2.
2. Allocates a new set of local registers for the called procedure.
3. Allocates a new frame on the procedure stack.
4. Changes the instruction pointer to point to the first instruction in the called procedure.
5. Stores the FP for the calling procedure in new local register r0 (PFP).
6. Stores the FP for the new frame in global register g15.
7. Allocates a save area for the new local registers in the new stack frame.
8. Stores the SP in new local register r1.

### Local Return Operation

On a return, the processor performs these operations:

1. Sets the FP in global register g15 to the value of the PFP in current local register r0.
2. Deallocates the current local registers for the procedure that initiated the return and switches to the local registers assigned to the procedure being returned to.

3. Deallocates the stack frame for the procedure that initiated the return.
4. Sets the IP to the value of the RIP in new local register r2.

The algorithms that the **call**, **callx**, and **ret** instructions use are described in greater detail in Chapter 9.

## PARAMETER PASSING

The processor supports two mechanisms for passing parameters between procedures: global registers and an argument list.

### Passing Parameters in Global Registers

The global registers provide the fastest method of passing parameters. Here, the calling procedure copies the parameters to be passed into global registers. The called procedure then copies the parameters (if necessary) out of the global registers after the call.

On a return, the called procedure can copy result parameters into global registers prior to the return, with the calling procedure copying them out of the global registers after the return.

### Passing Parameters in an Argument List

When more parameters need to be passed than will fit in the global registers, they can be placed in an argument list. This argument list can be stored anywhere in memory providing that the procedure being called has a pointer to the list. Commonly, a pointer to the argument list is placed in a global register.

Parameters can also be returned to the calling procedure through an argument list. Here again, a pointer to the argument is generally returned to the calling procedure through a global register.

The argument list method of passing parameters should be thought of as an escape mechanism and used only when there are not enough global registers available for passing parameters.

### Passing Parameters Through the Stack

A convenient place to store an argument list is in the stack frame for the calling procedure. Storing the argument list in the stack provides the benefit of having the list automatically deallocated upon returning from the procedure that set up the list. Space for the argument list is created by incrementing the SP, as described earlier in this chapter in the section titled "Stack Pointer."

Parameters can also be returned to the calling procedure through an argument list in the stack. However, care should be taken when doing this. The return argument list must not be placed in the frame for the called procedure, since this frame is deallocated on the return. Also, if the return list is to be placed in the frame of the calling procedure, the calling procedure must allocate space for this list prior to making the call.

## SYSTEM CALL

A system call is made using the call system instruction **calls**. This call is similar to a local call except that the processor gets the IP for the called procedure from a data structure called the *system-procedure table*. (System calls are sometimes referred to in this manual as "system-procedure-table calls.")

Figure 4-3 illustrates the use of the system-procedure table in a system call. The **calls** instruction requires a procedure-number operand. This procedure number provides an index into the system-procedure table, which contains IPs for specific procedures.

The system-call mechanism supports two types of procedure calls: local calls and supervisor calls. A local call is the same as that made with the **call** and **callx** instructions, except that the processor gets the IP of the called procedure from the system-procedure table. The supervisor call differs from the local call in two ways: (1) it causes the processor to switch to another stack (called the supervisor stack), and (2) it causes the processor to switch to a different execution mode.

The system call mechanism offers two benefits. First, it supports portability for application software. System calls are commonly used to call kernel services. By calling these services with a procedure number rather than a specific IP, applications software does not have to be changed each time the implementation of the kernel services is modified.

Second, the ability to switch to a different execution mode and stack allows kernel procedures and data to be insulated from applications code. This benefit is described in more detail later in this chapter in the section titled "User-Supervisor Protection Model".

## SYSTEM-PROCEDURE TABLE

The system-procedure table is a general structure, which the processor uses in two ways. The first way is as a place for storing IPs for kernel procedures, which can then be accessed through the system call mechanism. The processor gets a pointer to the system-procedure table from the initial memory image (IMI) as described in Chapter 3 in the section titled "System Data Structures."

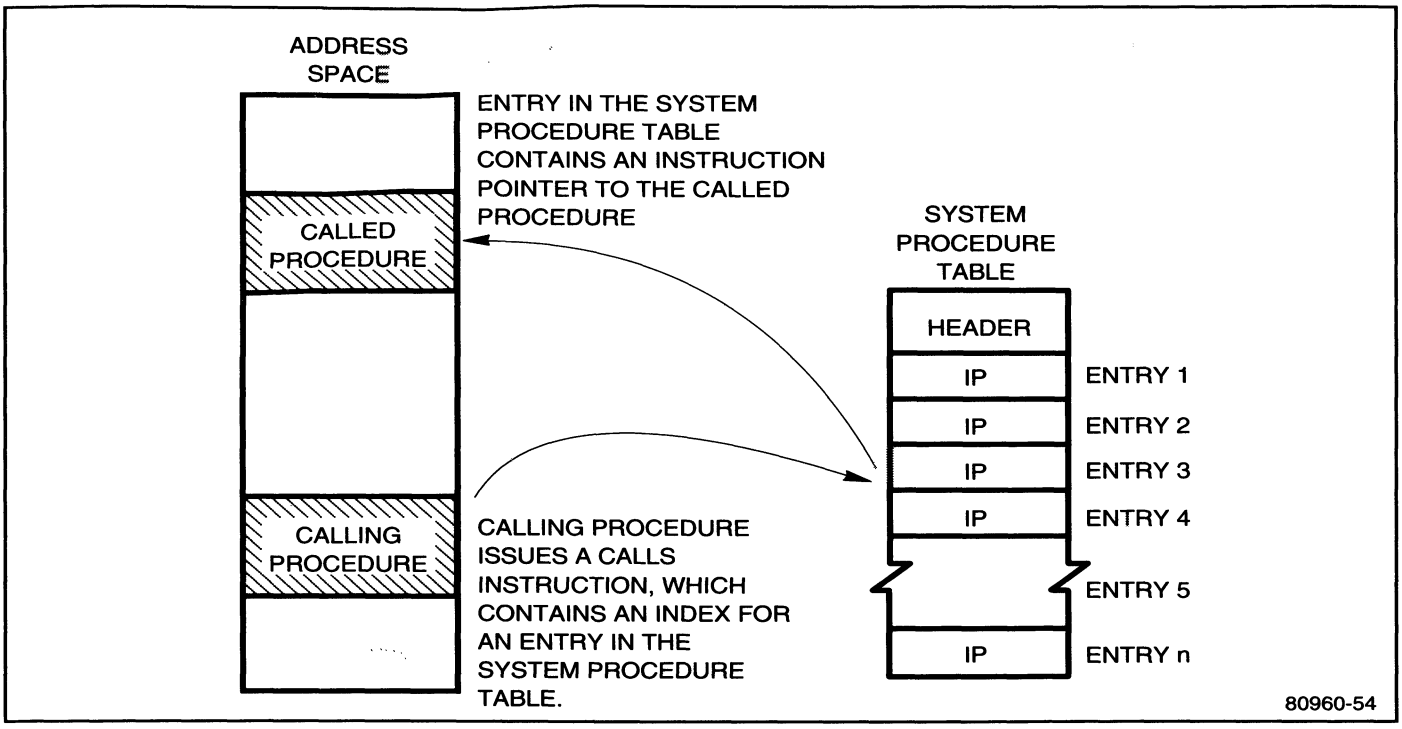


**Figure 4-3: System Call Mechanism**

The second way a system-procedure table is used is as a place for storing IPs for fault-handler procedures. Here, the processor gets a pointer to the system-procedure table from entries in the fault table, as described in Chapter 6 in the section titled "Fault-Table Entries."

The structure of the system-procedure table is shown in Figure 4-4. It is up to 1088 bytes in length and can have up to 260 procedure entries. The following sections describe the fields in this table.

The system procedure table may be located anywhere in memory on a SALIGN boundary. For more information on SALIGN see Appendix F, "SALIGN Parameters".

## Procedure Entries

The procedure entries specify the target IPs for the procedures that can be accessed through the system-procedure table. Each entry is one word in length and is made up of an address (or IP) field and a type field. The address field gives the address of the first instruction of the target procedure. Since all instructions are word aligned, only the 30 most-significant bits of the address are given. The processor automatically provides zeros for the least-significant bits. Entry 0 begins at byte 48 of the procedure table; the table can contain up to 260 entries.

The procedure entry type field indicates the type of call to execute: local or supervisor. The encodings of this field are shown in Table 4-2.

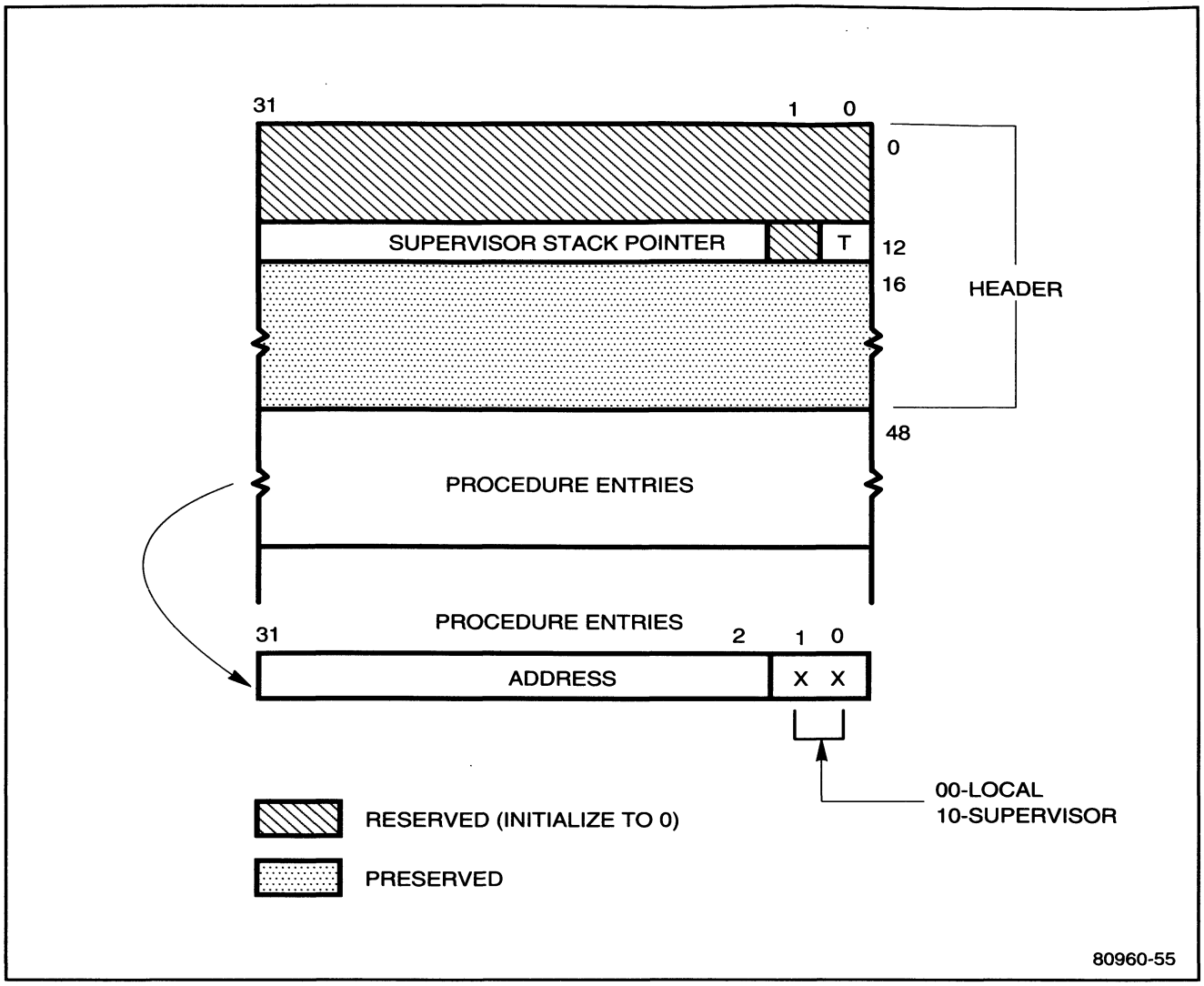


**Figure 4-4: System-Procedure Table Structure**

**Table 4-2: Encodings of Entry Type Field in System Procedure Table Entry**

| Entry Type Field | Procedure Type |
|---|---|
| 00 | local procedure |
| 01 | reserved |
| 10 | supervisor procedure |
| 11 | reserved |

### Supervisor Stack Pointer

When a supervisor call from user mode is made, the processor switches to a new stack called the *supervisor stack*. The processor gets a pointer to this stack from the supervisor-stack-pointer entry (bytes 12-15, bits 2-31) in the system-procedure table. Only the 30 most-significant bits of the supervisor stack pointer are given. The processor aligns this value to the next 64-byte boundary to determine the first byte of the new stack frame.

### Trace Control Flag

The trace-control flag (byte 12, bit 0) specifies the new value of the trace-enable flag when a supervisor call causes a switch from user mode to supervisor mode. Setting this flag to 1 enables tracing; setting it to 0 disables tracing. The use of this flag is described in the section in Chapter 7 titled "Trace Control on Supervisor Calls."

### System Call to a Local Procedure

When a **calls** instruction references a procedure entry designated as a local type ($00_2$), the processor executes a local call to the procedure selected from the system-procedure table. Neither a mode switch nor a stack switch occurs.

The **ret** instruction permits returns from either a local procedure or a supervisor procedure. The return status field in local register r0 determines the type of return action that the processor is to take. If the return status field is set to $000_2$, a local return is executed. In a local return, no stack or mode switching is carried out.

## USER-SUPERVISOR PROTECTION MODEL

The processor provides a mode and stack switching mechanism called the user-supervisor protection model. This protection model allows a system to be designed in which kernel code and data reside in the same address space as user code and data, but access to the kernel procedures (called supervisor procedures) is only allowed through a tightly controlled interface. This interface is provided by the system-procedure table.

The user-supervisor protection model also allows kernel procedures to be executed using a different stack (the supervisor stack) than is used to execute applications program procedures. The ability to switch stacks helps maintain the integrity of the kernel. For example, it would allow system debugging software or a system monitor to be accessed, even if an applications program crashes.

## User and Supervisor Modes

When using the user-supervisor protection model, the processor can be in either of two execution modes: user or supervisor. The difference between the two modes is that when in the supervisor mode, the processor:

- switches to the supervisor stack, and
- may execute a set of supervisor only instructions.

**NOTE**

In the 80960SA/SB implementation of the i960 architecture, the only supervisor-only instruction is the modify process controls instruction (**modpc**).

## Supervisor Calls

Mode switching between the user and supervisor execution modes is accomplished through a supervisor call. A supervisor call is a call executed with the **calls** instruction that references a supervisor procedure in the system-procedure table (i.e., a procedure with an entry type $10_2$).

When the processor is in the user mode and it executes a **calls** instruction, the processor performs the following actions:

- It switches to supervisor mode
- It switches to the supervisor stack
- It sets the return status field in register R0 of the calling procedure to $01X_2$, indicating that a mode and stack switch has occurred.

The processor remains in the supervisor mode until a return is performed from the procedure that caused the original mode switch. While in the supervisor mode, either the local call instructions (**call** and **callx**) or the **calls** instruction can be used to call supervisor procedures.

(The **call** and **callx** instructions call local (or user) procedures in user mode and supervisor procedures in supervisor mode. There is no stack or processor state switching associated with these instructions.)

When a **ret** instruction is executed and the return status field is set to $01X_2$, the processor performs a supervisor return. Here, the processor switches from the supervisor stack to the local stack, and the execution mode is switched from supervisor to user.

## Supervisor Stack

When using the user-supervisor mechanism, the processor maintains separate stacks in the address space, one for procedures executed in the user mode (local procedures) and another for procedures executed in the supervisor mode (supervisor procedures). When in the user mode, the local procedure stack (described at the beginning of this chapter) is used. When a supervisor call is made, the processor switches to the supervisor stack. It continues to use the supervisor stack until a return is made to the user mode.

The structure of the supervisor stack is identical to that of the local procedure stack (shown in Figure 4-2). The processor obtains the SP for the supervisor stack from the system-procedure table. When a supervisor call is executed while in the user mode (causing a switch to the supervisor stack), the processor aligns this SP to the next 64-byte boundary to form the new FP for the supervisor stack. When a local call or supervisor call is made while in the supervisor mode, the processor aligns the SP in the current frame of the supervisor stack to the next 64 byte boundary to form the FP pointer. This operation allows supervisor procedures to be called from supervisor procedures.

## Hints on Using the User-Supervisor Protection Model

The user-supervisor has two basic uses in an embedded system application:

1. to allow the **modpc** instruction to be used, and
2. to allow kernel code to use a separate stack from the applications code.

If an application does not require any of the above features, it can be designed to not use the user-supervisor protection model. Here, all procedure calls are to local procedures. If the system table is used, all the entries must be the local type (i.e., entry type $00_2$).

If access to the **modpc** instruction is required, but the other two features are not, it is suggested that the system be designed to always run in supervisor mode. At initialization, the processor automatically places itself in supervisor mode, prior to executing the first instruction. The processor then remains in supervisor mode indefinitely, as long as no action is taken to change the execution mode to user mode (i.e., using the **modpc** instruction to change the execution mode bit of the process controls to 0). With this technique, all of the procedure calling instructions (**call**, **callx**, and **calls**) can be used. The processor only uses one stack, which is considered the supervisor stack. It gets the supervisor stack pointer from local register r1. (Prior to making the first procedure call, the supervisor stack pointer must be loaded into r1.)

## BRANCH AND LINK

The **bal** (branch and link) and **balx** (branch and link extended) instructions provide an alternate method of making procedure calls. These instructions save the address of the next instruction (RIP) in a specified location, then branch to a target instruction or set of instructions. The state of the local registers and stack remains unchanged. (For the **bal** instruction, the RIP is automatically stored in global register g14; for the **balx** instruction, the location of the RIP is specified with one of the instruction operands.)

A return is accomplished with a **bx** (branch extended) instruction, where the address of the target instruction is the one saved with the branch and link instruction.

Branch and link procedure calls are recommended for calls to procedures that (1) do not call other procedures (i.e., for procedure calls that do not result in nesting of procedures) and (2) do not need many local variables (i.e., allocation of a new set of local registers does not provide any benefit). Here, local registers as well as global registers can be used for parameter passing.