HEWLETT ® PACKARD

PA-RISC 64-Bit Runtime Architecture Supplement

Version 3.3
October 1, 1997

Legal Notices

Introduction

PA-64 Runtime Supplement

Version 3.3
October 1, 1997

This document describes the specialized software conventions for the HP-UX operating system and the PA-RISC 2.0 architecture. The HP-UX 64-bit application programming model conforms to the common 64-bit software conventions for the PA-RISC 2.0 architecture, and the material presented here is supplementary to the common conventions, which are described in the separate document, *64-Bit Runtime Architecture for PA-RISC 2.0.*

While the document just mentioned covers the runtime architecture that is common to all operating environments on the PA-RISC 2.0 architecture, the chapters in this document cover material that is specific to the HP-UX operating system. Some of this material combines with the common conventions in contributing to the ABI, while other material describes implementation details below the level of the ABI.

The following topics are covered in this document:

❏ Program startup

❏ Kernel and embedded systems conventions

❏ Profiling

Program Startup

PA-64 Runtime Supplement

Version 3.3
October 1, 1997

This document covers the processor- and OS-specific details of program startup for 64-bit programs running under HP-UX on PA-RISC 2.0 processors. Program startup includes the following topics:

- ❏ Interface between the kernel loader (exec) and dld (for dynamically-bound programs) or crt0 (for statically-bound programs).
- ❏ Language-independent initializations performed by dld or crt0.
- ❏ Language-dependent initializations.
- ❏ Interface between dld or crt0 and the main program.
- ❏ Program termination.
- ❏ Symbols and variables defined by dld and crt0.
- ❏ Interface between the compiler drivers and the linker.

1. Background

Programs on HP-UX are always launched within an existing process by calling the exec system call. This call invokes the kernel loader, overlaying the running program with the new program. Typically, exec is called from another program immediately after spawning a new process with the fork system call.

The kernel loader recognizes three kinds of executable files:

- ❏ PA-32 SOM-format executables, whether statically-bound or “incomplete”. These programs are loaded as they always have been, and are not discussed further in this document.
- ❏ Statically-bound PA-64 ELF-format executables. These programs are loaded completely by the kernel loader, and control is transferred to the startup code linked in from the crt0 object module.
- ❏ Dynamically-bound PA-64 ELF-format executables. These programs are distinguished by the presence of a PT_INTERP record in the program header table. They are loaded by the kernel loader, but the “interpreter” designated by the PT_INTERP record is also loaded, and control is transferred to the interpreter. Typically, the interpreter is the dynamic loader, dld, although any statically-bound PA-64 ELF-format executable may be designated as an interpreter. Programs with an interpreter other than dld are not discussed further in this document (although the interface between exec and the interpreter remain as specified here).

The crt0 object file contains startup code that must be linked in to every statically-bound PA-64 program. This startup code contains the program entry point to which control is transferred from the kernel loader. The address of the startup code is recorded in the e_entry field of the ELF file header, and the kernel loader transfers control to this address after loading the program.

In a dynamically-bound program, the crt0 object file is not used, and all actions normally associated with crt0 are instead done by the dynamic loader, dld. The dynamic loader transfers control directly to the program's main program, function, or outer block, whose address is recorded in the e_entry field of the ELF file header.

2. Interface between the kernel loader and dld/crt0

When the kernel loader transfers control to either the crt0 startup code (for statically-bound programs) or the dynamic loader (for dynamically-bound programs), it sets up an initial stack and effectively makes a function call, passing certain information to the new program. The effective function prototype for the initial call from the kernel loader to the startup code (here called $START$) is as follows:

```
int $START$(
        int             argc,           /* number of command-line arguments */
        char            **argv,         /* pointers to command-line arguments */
        char            **envp,         /* pointers to environment strings */
        keybit_info_t   *keybits,       /* CPU information */
        load_info_t     *load_info      /* load information */
);
```

The structure keybit_info_t contains information about the processor currently executing the program. It has the following structure:

```
typedef struct keybit_info {
        int             cpu_version;    /* holds copy of cpu_version */
        int             fpu_info;       /* holds result of copr,0,0 */
        int             keycnt;         /* number of words of key bit info */
        unsigned int    key_bits[ ];    /* key bits returned from PDC */
} keybit_info_t;
```

The structure load_info_t contains load information. It has the following structure:

```
typedef struct load_info {
        uint64_t        li_version;             /* version number */
        uint64_t        li_length;              /* size of this structure */
        uint64_t        li_sysvec;              /* pointer to the sysvec table */
        uint64_t        li_aout_path;           /* pointer to a copy of a.out's path name */
        uint64_t        li_aout_hdr;            /* pointer to copy of a.out's ELF file header */
        uint64_t        li_aout_prog_hdr;       /* address of a.out's program header table */
        uint64_t        li_dld_taddr;           /* address of dld's text segment */
        uint64_t        li_dld_daddr;           /* address of dld's data segment */
} load_info_t;
```

The version and length fields are used to assure forward compatibility of executables on future releases of the operating system. Compatibility concerns dictate that existing fields of this structure cannot be moved or deleted; the structure may only be extended in such a manner that old executables may safely ignore the extra fields.

The sysvec table pointer is used by the program for making system calls. Its use is described below and in the separate chapter on system calls.

The kernel loader should copy the full path name of the original executable program (not the interpreter) to some location in the stack segment, and place a pointer to this (null-terminated) string in the li_aout_path field.

The final four fields of the load information structure are used only by dynamically-bound programs, and are for use by the dynamic loader.

The kernel loader must copy the ELF file header structure from the original executable program to some location in the stack segment, and place a pointer to it in the li_aout_header field.

The field li_aout_prog_hdr contains the virtual address of the original executable's program header table as it is mapped in memory. The program header table is identified by a separate PT_PHDR record in the program header table, but is always part of a loadable (PT_LOAD) segment. The kernel loader should provide the relocated value of the p_vaddr field from the PT_PHDR record.

The remaining two fields, li_dld_taddr and li_dld_daddr, provide the dynamic loader's own text and data segment addresses so dld can relocate itself.

3. Language-independent initializations

Both crt0 and the dynamic loader are responsible for the following initializations:

- ❏ Ensure correct alignment of the stack pointer
- ❏ Initializing the gp register
- ❏ Initializing the floating-point status register
- ❏ Initializing the pointer to the system call vector table
- ❏ Initializing the signal mechanism (if required)
- ❏ Initializing predefined system symbols
- ❏ Enabling profiling
- ❏ Initializing threads
- ❏ Invoking general initialization routines from .init sections
- ❏ Arranging for termination routines from .fini sections to be called at exit

These initializations should be performed in the order listed.

Stack pointer

The stack pointer must always be aligned to a 16-byte boundary, and the startup code should ensure this. (It's probably reasonable to make this a requirement of the kernel loader/startup code interface, so all the startup code needs to do is preserve the alignment.)

The gp register

The gp register needs to be initialized to its correct runtime value. In a statically-bound program, this value is determined at link time, and the linker sets the value of the __gp symbol. Thus, the startup code in crt0 simply needs to initialize the gp register with this constant value. In a dynamically-bound program, dynamic loader determines the gp values for each load module, and initializes all plabel descriptors with the proper values. The gp value for the main program is set automatically upon transfer of control to the main program, when dld makes a normal indirect call through the main program's plabel.

Floating-point status register

The floating-point status register may be set to an arbitrary initial value through the use of the +FP linker option when the main program is linked. The selected value is recorded in the executable file as the value of the symbol _FPU_STATUS. This should be an absolute symbol, with a type of STT_NOTYPE and a scope of STB_GLOBAL.

In a statically-bound program, crt0 will initialize the FP status register with the value of this symbol.

In a dynamically-bound program, dld will find the value of this symbol in the main program's symbol table, and initialize the FP status register.

System call vector table

The system call vector table is a table of pointers to kernel entry points, indexed by system call number.

In a statically-bound program, the system call stubs load the value of the variable __systab to get a pointer to this table. The startup code in crt0 must copy the value of the li_sysvec field in the load information structure to this variable.

In a dynamically-bound program, the system call stubs refer to the system call vector table directly, as the symbol __sysvec. The dynamic loader must define the value of this symbol, using the address in the li_sysvec field in the load information structure. The dynamic loader should also initialize the value of the variable __systab to ensure that references using the statically-bound technique also work correctly.

The system call mechanism is described in more detail in a separate chapter.

Signal mechanism

Currently, no special initialization is required for the signal mechanism.

Predefined system symbols

The linker, crt0, and dld all participate in the definition of a number of predefined system symbols available for application use. These symbols are described in Section 6, below.

Profiling

Profiling initializations are all done through entries in the .init section (described below). For more details, see the separate document, *Profiling 64-Bit Programs*.

Threads

Thread initializations to be performed depend on whether the program is linked with the pthreads library. If the program is linked with the pthreads library, an initialization routine in that library must be called to allocate thread-local storage for the main thread and to initialize its data structures. Otherwise, the program must still allocate thread-local storage for the main thread, but this storage does not need to include the extra data structures used by the pthreads library.

These initializations are done through the C library's initializer (using the .init section, described below). The C library initializer will check the global variable __thread_init, which will contain a function pointer referring to the thread initialization routine. Both the pthreads library and the C library will have definitions of this variable. If the program is linked with the pthreads library, the copy in the pthreads library will have precedence over that in the C library, and it point to a routine that performs the full pthreads initializations. Otherwise, the copy in the C library will be called, which points to a routine that performs only the basic initializations.

In statically-bound programs, the size of the thread-local storage segment is computed by the linker and stored in the global variable __tls_size, one of the predefined system symbols described in Section 6, below. In dynamically-bound programs, the dynamic loader computes the total amount of thread-local storage required by all load modules, and stores the sum in the __tls_size variable. This global variable is used by the thread initializer routine when it allocates the thread-local storage for the main thread, and by the thread creation routine to allocate storage for each new thread.

General initialization routines from .init sections

Each load module may contain a .init section that specifies initialization routine that must be called when the module is loaded. The .init section contains simply an array of function pointers, where each function pointer points to an initializer routine. When an application starts, the initializer routines for the main program and any startup DLLs must be called. When a DLL is loaded, the initializer routines for that DLL and any dependent DLLs loaded as a result must be called.

The .init sections may be constructed in three ways:

❏ Source code may contain a pragma that identifies a particular procedure as an initializer; the compiler will then add a function pointer to the .init section that refers to that procedure. This .init section will combine with others at link time, producing a single .init section with contributions from many different inputs.

❏ A compiler may generate an entry in the .init section automatically. For example, a C++ compiler is expected to create one function per translation unit with static constructors. It will place a pointer to this function in the .init section of that translation unit. This function will invoke the static constructors in the proper order during program startup.

❏ The linker's +I option (upper-case 'i') can be used to specify one or more procedures as initializers. The linker will add function pointers to the end of any existing .init section for each procedure specified in this manner, creating a .init section if necessary.

In a statically-bound program, the startup code in crt0 processes the single .init section in reverse order, using the predefined symbols __init_start and __init_end to identify the beginning and ending addresses of the section.

In a dynamically-bound program, the dynamic loader processes each load module in reverse order, checking for a .init section, which is identified by a DT_INIT record in the dynamic table. For each .init section found, it invokes the initializers in reverse order.

Initializers are invoked in reverse order so that dependent libraries are initialized before the libraries that depend on them. This allows the initializers for a library to invoke routines in dependent libraries with assurance that the dependent libraries have already been initialized. This property is used to ensure that the thread initialization routine, in the system threads library, is called before any application or user library initializers.

Initializers are parameterless functions with no return value.

Special initialization routines for the C library

In order to guarantee that the C library's initializers always run before any other library's initializers, the C library contains a single function pointer in a section named .preinit. The linker will construct a DT_PREINIT record in the dynamic table for any load module containing this section, and the dynamic loader will invoke the initializer(s) specified in this section prior to invoking initializers specified in .init sections.

Pre-initializers are parameterless functions with no return value.

There is no compiler support for building the .preinit section; it must be built in assembly language.

To ensure predictable behavior, there should be exactly one .preinit section in the C library, with a single function pointer. All of the C library's initializations should be invoked from this single initializer function.

Termination routines from .fini sections

Each load module may also contain a .fini section, with an array of function pointers referring to termination routines that are to be called when the program terminates via the exit routine. In a statically-bound program, the .fini section is delimited by the predefined symbols __fini_start and __fini_end; in a dynamically-bound program, it is identified by a DT_FINI record in the dynamic table. The startup code in crt0 and the dynamic loader arrange, via atexit, for these termination routines to be called when a program terminates. (The use of atexit for this purpose should not affect any implementation-imposed limit on the number of atexit routines that the application is allowed to register.)

Termination routines are invoked in forward order.

When a DLL is unloaded, its termination routines are also invoked at that time.

4. Language-dependent initializations

All language-dependent initializations are handled through the general initialization mechanism provided by the .init section, as described in the previous section. This assures that a program may contain a mixture of code from different compilers, and all appropriate initializations will be done for each language's runtime support library.

For Fortran, this means that the generated code for the main program should no longer include a call to the runtime library initialization routine. Instead, this routine should be made an initializer in the Fortran runtime library.

For C++, this means that the compiler should no longer generate an _main routine when it compiles main. Instead, the compiler should generate an initializer routine for each translation unit that has static constructors.

5. Interface between dld/crt0 and the main program

After all initializations have been performed, the startup code in crt0 or the dynamic loader will transfer control to the program's entry point, by making a call to the main program or outer block. The effective function prototype for this call is as follows:

```
int _start(
        int         argc,       /* number of command-line arguments */
        char        **argv,     /* pointers to command-line arguments */
        char        **envp,     /* pointers to environment strings */
);
```

In a statically-bound program, the program entry point is the startup code in crt0, and is named $START$. The linker sets the e_entry field of the ELF file header to the address of the startup code (unless the program is linked with the –e option, which specifies an alternate entry point). The main program or outer block is always named _start, and the startup code contains a direct reference to this symbol. The Fortran compiler should use this as the name of the main program. For C and C++, whose main program is always named main, the archive version of the C library contains a routine named _start, whose only action is to call main with the same three parameters as shown above.

In a dynamically-bound program, the address of the main program or outer block is recorded in the e_entry field of the ELF file header, and the dynamic loader transfers control to this address. The linker will set this field as follows:

- ❑ If the program is linked with the –e option, the named symbol will be recorded as the program's entry point.
- ❑ Otherwise, the linker will search for a symbol named _start. If this symbol is found, it is recorded as the program's entry point. This identifies a Fortran main program.
- ❑ Next, the linker will search for a symbol named main. If this symbol is found, it is recorded as the program's entry point. This identifies a C or C++ main program.
- ❑ Finally, if none of the above yield an entry point, the first address in the text segment is used as the program's entry point.

The program's entry point may reside in a DLL instead of the main program. If so, the linker will create an import stub in the main program for the entry point symbol (if one does not already exist), and set the entry point address to the address of that import stub.

There is no need for the _start routine in the DLL version of the C library.

The program may terminate either by calling exit directly, or by returning from the main program. In the latter case, the startup code will call exit, using the return value from the main program as the exit status.

6. Predefined system symbols

The standard programming environment provides a number of predefined system symbols for use by the application and communication between various components of the development environment. This section discusses the use of these symbols and how they are defined.

The linker defines the absolute symbols listed in Table 1. These symbols are defined when building a statically-bound or dynamically-bound executable, but not when building a DLL. The value of an absolute symbol may be obtained directly in a C program with the address-of operator (e.g., "&__SYSTEM_ID").

Table 1. Linker-defined absolute symbols

Symbol Name	Meaning
__SYSTEM_ID_D	Largest architecture revision level used by any compilation unit
_FPU_STATUS	Initial value of FPU status register
_end	Address of first byte following end of the main program's data segment; identifies the beginning of the heap segment
__TLS_SIZE_D	Size of TLS segment required by program

The C library defines the global variables listed in Table 2. An application may examine these variables to get information about the execution environment.

All of the symbols in Table 2 are statically initialized to the corresponding absolute symbols listed in Table 3, which are defined by the dynamic loader.For statically-bound programs, the linker will provide definitions of 0 for those symbols that do not also appear in Table 1; these symbols are then re-initialized by the startup code in crt0, which copies information passed from the kernel loader, as described in Section 2, into these variables. In dynamically-bound programs, the dynamic loader

Table 2. Predefined global variables

Symbol Name	Type	Meaning
_CPU_REVISION	long	Processor version level
_CPU_KEYBITS_1	unsigned long	Processor archictecture extension bits
_SYSTEM_ID	long	Highest architecture version required by program
_FPU_MODEL	long	FPU model
_FPU_REVISION	long	FPU version level
__argc	long	Copy of argument count
__argv	char **	Copy of argument vector
__envp, _environ	char **	Copy of environment vector
__tls_size	long	Total size of TLS segment
__load_info	load_info_t *	Pointer to load information structure

calculates the values of the symbols __SYSTEM_ID_D and __TLS_SIZE_D based on the DLLs loaded; it obtains the values of the remaining symbols from the information passed in from the kernel loader, as described in Section 2.

Table 3. Dynamic-loader-defined absolute symbols

Symbol Name	Meaning
__CPU_REVISION	Processor version level
__CPU_KEYBITS_1	Processor archictecture extension bits
__SYSTEM_ID_D	Highest architecture version required by program
__FPU_MODEL	FPU model
__FPU_REVISION	FPU version level
__ARGC	Argument count
__ARGV	Argument vector
__ENVP	Environment vector
__TLS_SIZE_D	Total size of TLS segments from all load modules
__LOAD_INFO	Address of load information structure

The initializations in the C library should each be in the form shown below:

```
extern char *__CPU_REVISION;
int _CPU_REVISION = &__CPU_REVISION;
```

This places an import record for the absolute symbol in the C library, and a dynamic relocation will direct the dynamic loader to write the load-time value of the absolute symbol into the global variable.

The dynamic loader should resolve references to these absolute symbols as if they were defined in a DLL at the end of the binding order.

In addition to the symbols listed in Table 1, the linker defines the symbols listed in Table 4, primarily for statically-bound programs. Their use in dynamically-bound programs is discouraged. Instead, the dlmodinfo routine in the dynamic loader library may be used to obtain similar information about each of the program's load modules.

Table 4. Predefined main program symbols

Symbol Name	Meaning
__text_start	Beginning of text segment
__text_start_f	Beginning of text segment, declared as a function
_etext	End of text segment
_etext_f	End of text segment, declared as a function
__data_start	Beginning of data segment
_edata	End of initialized data
__gp	Global pointer value
__init_start	Beginning of .init section
__init_end	End of .init section
__preinit_start	Beginning of .preinit section
__preinit_end	End of .preinit section
__fini_start	Beginning of .fini section
__fini_end	End of .fini section
__unwind_start	Beginning of unwind table
__unwind_end	End of unwind table

7. Interface between the compiler drivers and the linker

Because the crt0 file is used only for statically-bound programs, the compiler drivers will need to distinguish between static links and dynamic links. The compiler drivers should accept two new options, –noshared and –dynamic. The compiler drivers will pass these options on to the linker, which will use them to determine whether to link statically or dynamically. In addition, the compiler driver will include the crt0 file (/usr/lib/crt0.o) automatically at the beginning of the list of files for static links only. The default is to link dynamically, even if no DLLs are found at link time.

The C compiler driver should continue to pass the "–u main" option to the linker to ensure that a C main program will be loaded from an application archive library.

Conventions for Kernel and Embedded Systems

PA-64 Runtime Supplement

Version 3.3
October 1, 1997

This supplement discusses special coding conventions used for kernel development (and for development of other “embedded” applications and systems), and presents some example code sequences for various tasks. The code sequences shown in this chapter are intended to serve as guidelines and examples rather than as required coding conventions. These coding conventions may be selected with the +DCemb compiler option.

The kernel runtime model is based on the application model described in the common conventions document. The primary differences between the kernel model and the application model are:

- Embedded applications are standalone executables, and consists of a single load module. Therefore, all data may be treated as “own” data and may be accessed directly rather than through the linkage table.
- The gp register may be treated as a dedicated register that is never destroyed by a call; it does not need to be saved or restored anywhere.
- Function pointers are represented as a simple pointer to the function’s entry point. Thus, the code for materializing function pointers and for making indirect calls can be simplified to avoid the second level of indirection.
- Unless requested to generate relocatable code, the compiler may use absolute addresses in jump tables in the text segment.
- When programming for the kernel, the assembly-language programmer may have some special knowledge about the memory allocation of the code and data, and may write code that exploits this knowledge. For example, it may be known that the entire kernel is allocated within the first 4 Gigabytes of the address space, so address constants may be formed with the ADDIL/LDO pair familiar to 32-bit programmers.

1. Sample code sequences

1.1 Materializing function pointers

Function pointers may be obtained from the data segment, either as an initialized word or through the linkage table, as shown in the following examples. The TP' and P' operators, which generate pointers to function descriptors, are not used in the kernel model.

Function pointers may be obtained from the linkage table using the LT' and RT' operators:

```
            ADDIL     LT'function,gp              ; LTOFF21L
            LDD       RT'function(r1), tgt        ; LTOFF14DR
```

A function pointer may also be allocated explicitly in the data segment as a variable, and initialized directly with the address of the function:

```
    var:    .dword    function                    ; DIR64
```

1.2 Indirect procedure calls

In the kernel model, the extra level of indirection is removed from the indirect procedure call sequence:

```
            (assume function pointer is in register "fp")
            BVE,L     0(fp), r2
            <delay slot>
```

Profiling 64-Bit Programs

PA-64 Runtime Supplement

Version 3.3
October 1, 1997

This paper describes how *prof*- and *gprof*-style profiling work in the PA-64 runtime architecture.

Introduction

The HP-UX operating system provides two instrumentation techniques for profiling the execution of programs: program counter sampling, and procedure call counting. Program counter sampling, implemented within the OS at the clock interrupt level, provides statistical information about how much time is spent at different points in the program, enabling a programmer to determine what routines are responsible for most of the execution time. Procedure call counting, implemented by code added to each procedure at compile time, provides an exact count of the number of times each procedure was called.

These two techniques are usually employed together in conjunction with the *prof* and *gprof* tools. The –p (*prof*) and –G (*gprof*) compiler options have two effects:

❏ When used at compile time, they cause the compiler to generate code in each procedure that counts the calls;

❏ When used at link time, the compiler passes the options to the linker, which arranges for appropriate program startup code to enable program counter sampling and to write the profiling results to a data file when the program terminates.

An instrumented program accumulates the profiling information in a buffer while it is executing, then writes the data to a file when it terminates. The *prof* or *gprof* tool can then correlate this data to the symbol table of the a.out file and produce a profile report. These two tools differ in the amount of data collected and the detail provided in the report; *gprof* collects call-graph information, and allocates time spent in each procedure proportionally to the callers of that procedure (“chargeback”).

The support for profiling in the PA-32 runtime architecture involves the following components:

❏ The compilers. With the –p and –G options, the compilers instrument the compiled code by inserting calls to the _mcount routine.

❏ The startup files crt0.o, mcrt0.o, and gcrt0.o. Depending on link-time options, an application is linked with one of these startup files. If neither –p nor –G are specified, the application is linked with crt0.o, which contains a null _mcount routine (just to satisfy references from any code that was compiled with a profiling option). If –p is specified, the application is linked with mcrt0.o, which contains code to start profiling, a real version of _mcount, and code to write the accumulated profiling data to a file when the program terminates. If –G is specified, the application is linked with gcrt0.o, which contains similar code, but accumulates more data for use by gprof.

- ❑ The C library. It contains a *prof*-compatible version of the monitor routine. (The *gprof*-compatible version of monitor is contained in gcrt0.o.)
- ❑ The *prof* and *gprof* applications. These analysis tools read the data files produced by an instrumented applications, and generate the profile reports.

For the PA-64 runtime architecture, this new design eliminates the mcrt0.o and gcrt0.o startup files, moves the profiling-related code from those files into two new libraries (*libprof* and *libgprof*), and moves the standard version of monitor from the C library into *libprof*. In addition, it defines a new data file format to accomodate the 64-bit address space.

1. Program counter sampling: the profil system call

The profil system call is used to request the OS to start sampling or to stop sampling. When sampling is on, the OS samples the program counter (PC) at certain intervals (currently 100 times per second), maps the program counter to a “bucket” in the sample buffer, and increments the counter in that bucket. The parameters to profil control the location and size of the sample buffer, which is in user space, the starting address of the code region being profiled, and a scale factor, ranging from 0.0 to 1.0, that determines the mapping of PC values to the buckets. Larger scale factors provide greater granularity in the sampling data.

The profil system call has the following prototype:

```
int profil(
        unsigned short int *buff,
        size_t bufsiz,
        void *offset,
        unsigned int scale
);
```

The first argument, buff, is a pointer to the base of the sample buffer, and the second argument, bufsiz, is the size of this buffer, in bytes. The third argument, offset, is a pointer to the base of the text region to be profiled, and the fourth argument, scale, specifies the scale factor to be used in mapping locations in the text segment to buckets, which effectively defines the size of the text region.

The sample buffer is an array of 16-bit buckets, to which instructions in the text region are mapped according to the scale factor. The scale factor is a fixed-point fraction between 0.0 and 1.0, with an implied radix point 16 bits from the right. Thus, a pr_scale value of (1 << 16) specifies a scale factor of 1.0, where the size of the sample buffer is equal to the size of the text region. The largest meaningful scale factor is 0.5, represented as (1 << 15), since each bucket is half the size of an instruction.

The smallest allowed value of pr_scale is 2, corresponding to a scale factor of 1/32768. This results in one bucket for each 16,384 instructions.

For a given scale factor, the required size of the buffer is the size of the text region multiplied by the scale factor, or (text_size * pr_scale / 65536).

If scale is 0 or 1, sampling is turned off.

The third argument, offset, is expected to be the actual address of the beginning of the region of code to be profiled. This is different from a function pointer (which is expected by the monitor routine, described below). The C compiler provides a built-in function or macro that will convert a function pointer to the address of the code for that function. The function __get_entry(fp) will return the address of the code for the function represented by the function or function pointer fp.

2. Procedure call counting: the _mcount routine

When compiling code with the –p or –G option, the compiler instruments each procedure by inserting a call to _mcount at the beginning of each procedure (following the procedure prologue). This procedure has the following 64-bit interface:

```
void _mcount(
        unsigned long rp,
        unsigned long pc,
        unsigned int **counter_ptr
);
```

The first argument, rp, is a copy of the return pointer—that is, a pointer to the caller of the instrumented procedure. This gives *gprof* the call graph information it needs for chargeback.

The second argument, pc, is a pointer to an arbitrary instruction in the instrumented procedure itself. (This allows _mcount to determine its caller without using assembly language.)

The third argument, counter_ptr, is a pointer to a statically-allocated doubleword in the .sbss section. The compiler must allocate one of these doublewords for each instrumented procedure. When _mcount is called from a procedure for the first time, it will find *counter_ptr initialized to zero. It will then allocate a call counter in its own counter buffer, then store a pointer to that counter in *counter_ptr. On successive calls from that procedure, _mcount will find the pointer to the previously-allocated counter.

The following is the ideal PA-64 calling sequence for _mcount (not including the save and restore of the gp register):

```
copy     rp, arg0                       ; pass return pointer
addil    LR'counter_ptr_1, gp           ; form address of counter_ptr
ldo      RR'counter_ptr_1(r1), arg2
b,l      _mcount, rp                    ; call _mcount
copy     rp, arg1                       ; pass own pc
```

Note that the counter pointers must be 64-bit doublewords; in PA-32, they are 32-bit words. Otherwise, the calling sequence is the same as PA-32.

Counters should be 32 bit unsigned integers.

There are three versions of the _mcount routine:

- The C library contains an empty version, to satisfy references from code instrumented at compile time, but linked into a non-instrumented program.
- The *prof* library has a version that accumulates basic call counting information. For this version, the rp argument is not used, since prof does not do chargeback. When compiling with –p instead of –G, the compiler may choose to set this parameter to zero.
- The *gprof* library has a version that accumulates full call counting information.

3. The monitor library routine

The monitor routine is a higher-level interface to the profiling facility. It sets up program counter sampling, allocates data structures for both sampling and call counting, turns off profiling, and writes the profiling data to a disk file. It has the following interface:

```
void monitor(
        void (*lowpc)(),
        void (*highpc)(),
        long *buffer,
        int bufsize,
        int nfunc
);
```

The first two arguments define the range of addresses that will be profiled. These two arguments must be valid function pointers. For convenience in profiling the whole text segment in a statically-bound program, the linker defines two symbols, __text_start_f and __etext_f, which are the beginning and ending addresses, respectively, of the text segment. These symbols are declared in the header file <crt0.h>, and may be used as parameters to monitor.

The third and fourth arguments define the starting address and length of a buffer for the sample buckets, and the final argument is the total number of procedure call counters that should be allocated.

The monitor routine will automatically calculate the proper scale factor and call the profil system call to enable sampling. It will also allocate an array of procedure call counters and the data structures necessary for _mcount to allocate counters from this array dynamically.

There are two versions of the monitor routine:

- The *prof* library has a version that allocates data structures for basic call counting information and dumps the data in a *prof*-compatible format.
- The *gprof* library has a version that allocates data structures for full call counting information and dumps the data in a *gprof*-compatible format.

4. Linking for use with prof and grof

When linking a program for use with *prof*, the compiler driver will pass the "–lprof" (lower-case 'L') option to the linker. The –l option causes the linker to load the *prof* library before the C library, where it will find *prof* versions of the monitor and _mcount routines.

When linking a program for use with *gprof*, the compiler driver will pass the "–lgprof" option to the linker. The –l option causes the linker to load the *gprof* library before the C library, where it will find *gprof* versions of the monitor and _mcount routines.

The *prof* and *gprof* libraries may be delivered in shared and archive forms. The shared forms are normal DLLs, and will be loaded automatically when the program is executed. The archive forms will not be actual archive libraries, but instead will be relocatable object files named with a ".a" suffix. This allows them to be loaded with the -l linker option, but ensures that they will be loaded into the program. If packaged as a real archive library, the linker would not load any members of the library that are not directly referenced from the program.

When linking a program for either profiler tool, the compiler driver will also pass the "–L /usr/ccs/lib/libp" option to the linker. This option adds an extra directory to the library search path, where instrumented versions of selected system libraries may be found. These libraries will have been compiled with the –p option, so that they are instrumented for procedure call counting.

Note that the linker does not support the –p and –G options for profiling. These options are meaningful only to the compiler drivers, which in turn pass the appropriate options to the linker. To invoke the linker directly, the linker options mentioned above must be used instead.

Note also that no separate crt0.o file is required.

5. Running the instrumented program

When an instrumented program starts execution, the initializer in the .init section of the profiling library (*libprof* or *libgprof*) is called before control reaches main. The initializer will call monitor to start profiling on the program's text segment; allocate the data structures used for sampling and call counting; and arrange, via atexit, to call monitor again to stop profiling at the end of the program.

As the program is running, program counter sampling data will be accumulated (by the OS) in the buffer allocated for it. Any code compiled with instrumentation will call the _mcount routine each time it is called to accumulate the call counting data.

When an instrumented program terminates, one of its *atexit* actions will be to call monitor to disable profiling and to write the profiling data to a disk file.

6. Profile data file format

The existing data file formats used by *prof* and *gprof* are designed for a 32-bit address space, do not provide for extensions, and do not identify themselves. The new 64-bit data file format described here is a simple unified format that can support both *prof* and *gprof*, with enough header information to identify the structure of the file. It can also support future extensions for larger sample buckets and counters, and is capable of containing other types of profile data for profiling tools other than *prof* and *gprof*.

The *prof* and *gprof* programs must be able to distinguish an old-format 32-bit data file from a 64-bit data file. Since the old format does not contain any reliable identification, the utilities must assume that a data file that does not contain a 64-bit file header is an old-format file. The utilities should also ensure that a 32-bit data file is associated with a 32-bit program file (SOM format), and that a 64-bit data file is associated with a 64-bit program file (ELF format).

A profile data file consists of a fixed file header followed by an arbitrary number of data sections. The file header identifies the file as a profile data file. Each data section begins with a section header that identifies the contents and structure of the data in that section.

The file header is eight bytes long. The first seven bytes contain the characters "<PROF1>", and the eighth byte contains a newline character. This header identifies the file as Version 1 of the profile data file format.

Each section header contains, at a minimum, two doublewords. The first doubleword contains a section type; section types used for *prof* and *gprof* are defined below. The second doubleword contains the length of the section (including the section header). Depending on the section type, additional fields may be defined.

The following three section types are defined for *prof* and *gprof*:

PROF_SECT_SAMPLES_64

This type of section contains a program counter sample buffer. The section header contains additional fields defining the low and high pc values for the text segment, the scale factor (as passed to the profil system call), and the size of each bucket (currently, 2 bytes). This section type is used for both *prof* and *gprof*.

PROF_SECT_CALL_COUNTS_64

This type of section contains call counters. The section header contains two additional fields defining the number of counters and the size of each counter (currently, 4 bytes). The data following the section header is in two parts: an array of doublewords containing the pc values corresponding to each counter, followed by the array of counters (whose size is identified in the section header). This section type is used only by *prof*.

PROF_SECT_CALL_ARCS_64

This type of section contains call arc counters. The section header contains two additional fields defining the number of arcs and the size of each counter (currently, 4 bytes). The data following the section header is in two parts: an array of 16-byte arc structures, followed by the array of counters (whose size is identified in the section header). An arc structure contains two doublewords: a *from* pc, and a *to* pc. This section type is used only by *gprof*.

C declarations for the section headers for these three section types are shown below.

```
#define PROF_SECT_SAMPLES_64            1
#define PROF_SECT_CALL_COUNTS_64        2
#define PROF_SECT_CALL_ARCS_64          3

struct prof_secthdr_samples {
        long sect_type;                 /* PROF_SECT_SAMPLES_64 */
        long sect_size;                 /* size of section, including header */
        unsigned long lowpc;            /* low pc for text region */
        unsigned long highpc;           /* high pc for text region */
        int scale;                      /* scale factor */
        int entry_size;                 /* size of each sample bucket = 2 */
};

struct prof_secthdr_call_counts {
        long sect_type;                 /* PROF_SECT_CALL_COUNTS_64 */
        long sect_size;                 /* size of section, including header */
        int ncounters;                  /* number of counters */
        int counter_size;               /* size of each counter = 4 */
};

struct prof_secthdr_call_arcs {
        long sect_type;                 /* PROF_SECT_CALL_ARCS_64 */
        long sect_size;                 /* size of section, including header */
        int narcs;                      /* number of arcs */
        int counter_size;               /* size of each counter = 4 */
};
```