UNWIND PA64 Functional Specification.

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with furnishing, performance, or use of this material. Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Version 1.8
Sept. 16,1997

HEWLETT PACKARD

California Language Laboratroy
11000 Wolfe Road
Cupertino, California 95014
Last Printing: October 8, 1997

1.0 Data Structures

- FYI: scalar type definitions

```
unsigned long   general_reg;

unsigned int bit32;

int boolean;
```

- frame record structures

```
typedef struct
    {
        unsigned long   size;
        general_reg     sp;
        general_reg     return_link_offset;
        general_reg     gp;             /* the global pointer value associated
                                           with a given shared library */
        general_reg     rp;
        general_reg     mrp;
        general_reg     r3
        general_reg     r4
        unsigned long   reserved[4]
    } curr_frame_info;

typedef struct
    {
        unsigned long       size;
        general_reg         sp;
        general_reg         return_link_offset;
        general_reg         gp;
        uw_rec_def          uw_rec;
        long                uw_index;
        general_reg         r3
        general_reg         r4
        unsigned long       reserved[4]
    } prev_frame_info;
```

- unwind descriptors

```
typedef struct {
    unsigned int no_unwind:1;                   /* 0..0 */
    unsigned int is_millicode:1;                /* 1..1 */
    unsigned int reserved0:1;                   /* 2..2 */
    unsigned int region_descr:2;                /* 3..4 */
    unsigned int reserved1:1;                   /* 5..5 */
    unsigned int entry_sr:1;                    /* 6..6 */
    unsigned int entry_fr:4;                    /* 7..10*/
    unsigned int entry_gr:5;                    /* 11..15*/
    unsigned int args_stored:1;                 /* 16..16*/
    unsigned int reserved2:3;                   /* 17..19*/
    unsigned int stk_overflow_chk:1;            /* 20..20*/
    unsigned int two_inst_sp_inc:1;             /* 21..21*/
    unsigned int reserved3:1;                   /* 22 */
    unsigned int c_plus_cleanup:1;              /* 23 */
    unsigned int c_plus_try_catch:1;            /* 24 */
    unsigned int sched_entry_seq:1;             /* 25 */
    unsigned int reserved4:1;                  /* 26 */
    unsigned int save_sp:1;                    /* 27..27*/
    unsigned int save_rp:1;                    /* 28..28*/
    unsigned int save_mrp:1;                   /* 29..29*/
    unsigned int reserved5:1;                  /* 30..30*/
    unsigned int has_cleanup:1;                /* 31..31*/
    unsigned int reserved6:1;                  /* 32..32*/
    unsigned int is_HPUX_int_mrkr:1;           /* 33..33*/
    unsigned int large_frame_r3:1;             /* 34..34*/
    unsigned int alloca_frame:1;               /* 35 */
    unsigned int reserved7:1;                  /* 36..36*/
    unsigned int frame_size:27;                /* 37..63*/
} descriptor_bits;

typedef struct  { /* unwind entry as the unwind library stores it in the prev frame record */
    descriptor_bits unwind_descriptor_bits;
    bit32 region_start_address;
    bit32 region_end_address;
} uw_rec_def;


typedef struct {
    unsigned long table_start;    /* Start address of a table, e.g. the unwind table */
    unsinged long table_end;      /* End address of same table */
} table_record;

typedef struct {
    double          so_fp12;
    double          so_fp13;
    double          so_fp14;
    double          so_fp15;
    double          so_fp16;
    double          so_fp17;
    double          so_fp18;
    double          so_fp19;
    double          so_fp20;
    double          so_fp21;

    unsigned long       so_rp;
    unsigned long       so_sp;
    unsigned long       so_mrp;  /* gr31 */

    unsigned long       so_gr3;
    unsigned long       so_gr4;
    unsigned long       so_gr5;
    unsigned long       so_gr6;
    unsigned long       so_gr7;
    unsigned long       so_gr8;
    unsigned long       so_gr9;
    unsigned long       so_gr10;
    unsigned long       so_gr11;
    unsigned long       so_gr12;
    unsigned long       so_gr13;
    unsigned long       so_gr14;
```

```
        unsigned long       so_gr15;
        unsigned long       so_gr16;
        unsigned long       so_gr17;
        unsigned long       so_gr18;
    } state_vec;

    typedef struct {
        int   bit;
        int   error_code;
    } usertrap_info;

    typedef struct {
        int   status;
        int   operation;
        int   op_class;
        int   format;
        int   format_src;
        int   reg_src1;
        int   reg_src2;
        int   reg_dest;
    } ieee_info_rec;

    typedef struct /* unwind_entry_rec as stored in the unwind table by the linker */ {
        bit32 lo, hi;               /* These are offsets from text base address */
        descriptor_bits unwind_descriptor_bits;
    } unwind_entry_rec;
```

2.0 Function definitions

- **void U_init_frame_record(curr_frame_info *frame)** -- Fills in the record, “*frame*” with a description of the stack frame for *U_init_frame_record()* and some register values that are followed during the process of unwinding the processor stack. Table 1.0 describes what values are placed in the fields of “*frame*.”

record field	**value assigned**
size	0
sp	contents of %sp (gr30)
return_link_offset	pc value during execution of U_init_frame_record
mrp	0
r3	contents of %r3
r4	contents of %r4
reserved[4]	(not assigned)

- **int U_get_previous_frame(curr_frame_info *curr_frame, prev_frame_info *prev_frame)** Upon entry, “curr_frame_info” contains:

curr_frame	
record field	**value contained upon entry**
size	size of current frame, also referred to as the “callee’s frame.”
sp	The current frame’s %sp value. (that is the Top of Stack while control was executing the object code that created the current frame)

curr_frame **record field**	**value contained upon entry**
return_link_offset	The return link address into the “caller procedure”. By definition, it is O.K. for this value to point to an export stub. The unwind library will consult stub tables to update this field to point to the actual return point in the “callee procedure.” At this time, the PA64 run time architecture definition does not allow for Export stubs. Thus, this situation will only be noticed in PA32.
mrp	NA
r3	the callee’s %r3 value
r4	the callee’s %r4 value
reserved[4]	NA

Upon exit, “curr_frame_info” contains

curr_frame **record field**	**value contained upon return**
size	(unchanged) size of current frame, also referred to as the “callee’s frame.”
sp	(unchanged) The current frame’s %sp value. (that is the Top of Stack while control was executing the object code that created the current frame)
return_link_offset	The actual return point in the “callee procedure.”
mrp	NA
r3	(unchanged) the callee’s %r3 value
r4	(unchanged) the callee’s %r4 value
reserved[4]	NA

and the “prev_frame_info” record contains information regarding the previous frame (that belonging to the caller):

prev_frame **record field**	**value contained upon return**
size	size of the previous frame, also referred to as the “caller’s frame.”
sp	The previous frame’s %sp value. (that is the Top of Stack while control was executing the object code that created the previous frame)
return_link_offset	The return link address into the procedure which called the “caller procedure”. Once again, by definition, it is O.K. for this value to point to an export stub. At this time, the PA64 run time architecture definition does not allow for Export stubs. Thus, this situation will only be noticed in PA32.
uw_rec	unwind records for the caller procedure
uw_index	index of the unwind table entry. (0..N-1, where N is the number of entries in the table)
r3	the caller’s %r3 value
r4	the caller’s %r4 value
reserved[4]	NA

In the most simple case (no interrupts or stubs), the “previous frame” is the frame of the “caller” procedure that called the “callee” procedure whose frame is described by **curr_frame*. In some cases, control flow had reached the “callee” procedure via an HP_UX interrupt in which case the stack contains an interrupt marker (called sig_context which contains the saved system state) and the “callee” procedure is a user space interrupt handler (in HP_UX, it is *_sigreturn*). By referring to the information in the interrupt marker, U_get_previous_frame will calculate which routine was interrupted and fill in the "previous_frame" record with a description of the interrupted routines stack frame.

- **table_record U_get_shLib_unw_tbl(address key)** -- Delivers the start address and the end address for a shared library unwind table. If the input parameter *key* does not point to an address (instruction or data) within a loaded shared library, **U_get_shLib_unw_tbl(address key)** returns -1 in the *table_record.table_start* field, else it returns the start and end addresses for the shared library unwind table.

- **void U_update_state_vector(struct statevec *state_vec**
  prev_frame_info *previous_frame_info,
  address uw_start_adr,
  address uw_end_adr,
  address return_link_offset)

Throughout this semantic description of *U_update_state_vector*, we shall refer to the procedure whose %sp, %r3 and %r4 values are passed in via the *previous_frame_info* parameter as the "caller." The procedure it called shall be referred to as the "callee" or "current procedure."

Given:

- *state_vec* -- A pointer to a *state_vec* record containing non-scratch (callee saves) register values at the moment control flow entered the "callee procedure."

- *previous_frame_info* -- A pointer to a *prev_frame_info* record containing the frame size, the sp, r3, and r4 values and the unwind table entries for the "caller" procedure.

- *uw_start_adr, uw_end_adr* -- the unwind region start and end addresses for the "caller" procedure who's stack state is described by "previous_frame". (Note these are the start and end of the unwind region (in code space.) Not the location of the unwind entry in the unwind table. A common user error is to confuse these two)

- *return_link_offset* the "return link offset" to the "caller procedure" (who's stack state is described by "prev_frame")

U_update_state_vector() restores the non-scratch general and floating point register values in the *state_vec* to the values the registers contained when control flow entered the "caller" (previous) procedure.

- **void U_resume_execution(struct statevec *state_vec, address resume_at_pc, address resume_at_gp)**

Given:

- *statevec* -- A pointer to a *state_vec* record containing non-scratch (caller saves) register values.

- *resume_at_pc* -- An instruction address in a procedure whose callee-saves register values are stored in *statevec*

- *resume_at_gp* -- The gp value for the code at address, "resume_at_pc."

Partially sets the system's processor state to the state described by the state vector, then branches to the address indicated by *resume_at_pc*. *U_resume_execution()* requires that the information in the state vector and *resume_at_pc* address be obtained from a "context preserving" unwind process and that the context described by the contents of the state vector and by *resume_at_pc* still have a frame on the procedure call stack. Note: that the entire system state is not (and cannot) be restored by the Unwind library. Any values the procedure kept in “caller saves” registers cannot be restored by the unwind library. “Resume_at_gp” can be obtained from prev_frame_info->gp after a call to U_get_previous_frame.

- **table_record U_get_unwind_table() --** returns a record containing the 64 bit address of the unwind table start and the64 bit address of the end of the unwind table. By definition, end of the unwind table is the address of the first byte after the last entry in the unwind table (e.g. unwind end does not point to the last entry of the unwind table.)
- **address U_get_shLib_text_addr(address key)**-- Given an address of an instruction or data item with a currently loaded shared library,*U_get_shLib_text_addr()* returns the 64 bit text address of the shared library. Unwind entries in the shared library unwind table are offsets from this text address. Returns -1 if the dynamic loader is not loaded or the key is not an address within a shared library.
- **address U_get_unwind_entry(general_reg program_counter,**
  general_reg utab_start, /* addr where unwind table starts */
  general_reg utab_end) /* addr where unwind table ends */ -- returns a pointer to the unwind table entry for the code segment containing the *program_counter* address. Note: address is typedefined as unsigned long -- a 64 bit quantity in pa64. (32 bits in pa32)
- **void U_init_frame_record(curr_frame_info* start_frame)** -- Initializes the fields in *start_frame* so it describes the stack frame used by *U_init_frame_record().* There is one exception: The *return_link_offset* field of *start_frame* reflects a pc_offset within *U_init_frame_record().* A call to *U_prep_frame_rec_for_unwind(start_frame)* will set the *return_link_offset* field to the return link offset value as required by U_get_previous_frame.
- **void U_prep_frame_rec_for_unwind(curr_frame_info* cfi)** -- Fills in *cfi*’s *return_link_offset* field with the return pointer to the caller of the routine whose frame is described by *cfi*.
- **void U_get_my_context(curr_frame_info* start_frame, struct statevec * state_vec)** -- Initializes the fields in *start_frame* and the fields in *state_vec* to describe the processor state during the execution of *U_get_my_context.* This is the method for initializing a context restoring stack unwind which has the following basic form exhibited by the following ANSI C source excerpt:

```
state_vec state_vector; /* State vector */
prev_frame_info previous_frame;
curr_frame_info current_frame;
unsigned long adjustment;
U_get_my_context(&current_frame, &state_vec);
U_prep_frame_rec_for_unwind(&current_frame);
while(!termination_condition) {
    U_get_previous_frame(&current_frame,&previous_frame);
    if (resume_to_user_code_condition_has_been_met) {
        U_resume_execution(&state_vec, current_frame.return_link_offset);
        /* Note: U_resume_execution returns the control of flow to the user’s code. Control flow. */
        /* never reaches this point */
    }
    adjustment = U_get_shLib_text_addr(current_frame.return_link_offset);
    if (adjustment == -1)
        adjustment = 0;
/* Adjust current_rlo if it is an absolute address addressing
* a location in a shared library. The “unwind start” and “unwind
* end” values for shared libraries are offsets from the start
* of the shared library’s text space. Thus we must subtract the
* absolute starting address of the text space of the shared library
* from current_rlo.
*/
```

```
    U_update_state_vector(&state_vec, &previous_frame,
                          prev_fr.uw_rec.boundaries.start,
                          prev_fr.uw_rec.boundaries.end,
                          curr_fr.pc_offset - (unsigned int) adjustment);
    /* copy pertinent fields from the previous frame record to the next loop iteration's current fr */
    U_copy_frame_info((&current_frame,&previous_frame);
}
```

- **void U_copy_frame_info(curr_frame_info *current, prev_frame_info *previous) --** The size, sp, pc_offset, r3, and r4 fields are copied from *previous* to *current.*

- **curr_frame_info U_get_current_frame() --** Returns a *curr_frame_info* structure which describes the stack frame of the routine that called *U_get_current_frame*() The *curr_frame_info* structure returned is ready for use in calling *U_get_previous_frame.* This routine is good for initializing a non-context restoring unwind.

- **int U_is_stack_unwound(address sp, unsigned int uw_desc_wd1,unsigned int uw_desc_wd2) --** Returns 1 if the stack is fully unwound. Returns 0 otherwise.

- **void U_TRACEBACK(int sig_number, struct sigcontext* ptr)** -- Displays the error status followed by a stack trace. The first parameter, “sig_number” is used to select which of about 19 error messages to print as listed here. The format of the stack trace is the same as that described under *U_STACK_TRACE().*

message
Signal 1: hangup
Signal 2: interrupt
Signal 3: quit
Signal 4: illegal instruction
Signal 5: trace trap
Signal 6: abort
Signal 7: not enough memory available
Signal 8: floating point exception
Signal 9: kill
Signal 10: bus error
Signal 11: segmentation violation
Signal 12: bad argument for system call
Signal 13: write on a pipe with no one to read
Signal 14: alarm clock trap
Signal 15: software termination signal
Signal 16: user defined signal 1 trap
Signal 17: user defined signal 2 trap
Signal 18: death of a child
Signal 19: power fail

- **boolean U_IS_MILLI_CODE(general_reg pc) --** returns 1 if the instruction address in *pc* is pointing into a millicode routine; else returns 0.

- **void U_STACK_TRACE()** -- delivers a stack trace to stderr. The trace display begins with the function which called *U_STACK_TRACE()* and concludes with the executable’s “start” code (typically found in crt0.o or in libc.sl). The fields of the stack trace is as follows:

field	**contents**	**format**
1st	the depth (counted in user code stack frames excluding stubs and interrupt markers) of the current procedures frame on the run time stack.	(decimal integer)
2nd	return link address where control will return to this function when it’s callee executes a “return” (such as a bv 0(rp)).”	hex
3rd	same address as field #2 with symbol information. The symbol information will not be provided if the symbols have been stripped from the “a.out” file.	procedure label + hex offset
4th	Name of the load module in which the procedure resides.	[HP_UX path name]

Example:

```
( 0)  0x000031f4   foo + 0x14  [./a.out]
( 1)  0x00003214   bar + 0x14  [./a.out]
( 2)  0x0000323c   main + 0x14  [./a.out]
( 3)  0xc0046e98   _start + 0xa8  [/usr/lib/libc.2]
( 4)  0x00002730   $START$ + 0x160  [./a.out]
```

- **void U_TRAP_STACK_TRACE(curr_frame_info start_frame)** -- delivers a stack trace to stderr. The trace display begins with the frame represented by the fields in *start_frame*. The fields and their contents are the same as those described for *U_STACK_TRACE()*.
- **int U_NextFrame(curr_frame_info frame_rec)** -- Performs an unwind step, filling in the fields of *frame_rec()* with information describing the next deeper user code frame on the stack. Returns 0 if the unwind step was successful. Prints the message, “Stack_Trace: error while unwind stack,” and returns -1 if the unwind step was not successful.

2.1 Changes from the PA32 interface

1. Many functions which were defined with *integer* return values in pa32 but which returned nothing, have been specified as having *void* return values in the pa64 interface.