NAME

link - dynamic loader and link editor interface

SYNOPSIS

#include <sys/types.h>
#include <nlist.h>
#include <link.h>

DESCRIPTION

The include file <*link.h*> declares several structures that are present in dynamically linked programs and libraries. The structures define the interface between several components of the link-editor and loader mechanism. The layout of a number of these structures within the binaries resembles the a.out format in many places as it serves such similar functions as symbol definitions (including the accompanying string table) and relocation records needed to resolve references to external entities. It also records a number of data structures unique to the dynamic loading and linking process. These include references to other objects that are required to complete the link-editing process and indirection tables to facilitate *Position Independent Code* (PIC for short) to improve sharing of code pages among different processes. The collection of data structures described here will be referred to as the *Run-time Relocation Section (RRS)* and is embedded in the standard text and data segments of the dynamically linked program or shared object image as the existing a.out(5) format offers no room for it elsewhere.

Several utilities cooperate to ensure that the task of getting a program ready to run can complete successfully in a way that optimizes the use of system resources. The compiler emits PIC code from which shared libraries can be built by ld(1). The compiler also includes size information of any initialized data items through the .size assembler directive. PIC code differs from conventional code in that it accesses data variables through an indirection table, the Global Offset Table, by convention accessible by the reserved name _GLOBAL_OFFSET_TABLE_. The exact mechanism used for this is machine dependent, usually a machine register is reserved for the purpose. The rational behind this construct is to generate code that is independent of the actual load address. Only the values contained in the Global Offset Table may need updating at run-time depending on the load addresses of the various shared objects in the address space.

Likewise, procedure calls to globally defined functions are redirected through the Procedure Linkage Table (PLT) residing in the data segment of the core image. Again, this is done to avoid run-time modifications to the text segment.

The linker-editor allocates the Global Offset Table and Procedure Linkage Table when combining PIC object files into an image suitable for mapping into the process address space. It also collects all symbols that may be needed by the run-time link-editor and stores these along with the image’s text and data bits. Another reserved symbol, *_DYNAMIC* is used to indicate the presence of the run-time linker

structures. Whenever _DYNAMIC is relocated to 0, there is no need to invoke the run-time link-editor. If this symbol is non-zero, it points at a data structure from which the location of the necessary relocation- and symbol information can be derived. This is most notably used by the start-up module, *crt0*. The _DYNAMIC structure is conventionally located at the start of the data segment of the image to which it pertains.

DATA STRUCTURES

The data structures supporting dynamic linking and run-time relocation reside both in the text and data segments of the image they apply to. The text segments contain read-only data such as symbols descriptions and names, while the data segments contain the tables that need to be modified by during the relocation process.

The _DYNAMIC symbol references a *_dynamic* structure:

```
struct  _dynamic {
        int         d_version;
        struct      so_debug *d_debug;
        union {
                    struct section_dispatch_table *d_sdt;
        } d_un;
        struct  ld_entry *d_entry;
};
```

d_version This field provides for different versions of the dynamic linking implementation. The current version numbers understood by ld(1) and ld.so(1) are *LD_VERSION_SUN (3)*, which is used by the SunOS 4.x releases, and *LD_VERSION_BSD (8)*, which has been in use since FreeBSD 1.1.

d_un Refers to a *d_version* dependent data structure.

so_debug this field provides debuggers with a hook to access symbol tables of shared objects loaded as a result of the actions of the run-time link-editor.

The *section_dispatch_table* structure is the main "dispatcher" table, containing offsets into the image's segments where various symbol and relocation information is located.

```
struct section_dispatch_table {
        struct      so_map *sdt_loaded;
        long        sdt_sods;
        long        sdt_filler1;
```

```
        long    sdt_got;
        long    sdt_plt;
        long    sdt_rel;
        long    sdt_hash;
        long    sdt_nzlist;
        long    sdt_filler2;
        long    sdt_buckets;
        long    sdt_strings;
        long    sdt_str_sz;
        long    sdt_text_sz;
        long    sdt_plt_sz;
};
```

sdt_loaded
A pointer to the first link map loaded (see below). This field is set by **ld.so**

sdt_sods The start of a (linked) list of shared object descriptors needed by *this* object.

sdt_filler1 Deprecated (used by SunOS to specify library search rules).

sdt_got The location of the Global Offset Table within this image.

sdt_plt The location of the Procedure Linkage Table within this image.

sdt_rel The location of an array of *relocation_info* structures (see a.out(5)) specifying run-time relocations.

sdt_hash The location of the hash table for fast symbol lookup in this object's symbol table.

sdt_nzlist The location of the symbol table.

sdt_filler2 Currently unused.

sdt_buckets
The number of buckets in *sdt_hash*

sdt_strings
The location of the symbol string table that goes with *sdt_nzlist*.

sdt_str_sz The size of the string table.

sdt_text_sz
The size of the object's text segment.

sdt_plt_sz The size of the Procedure Linkage Table.

A *sod* structure describes a shared object that is needed to complete the link edit process of the object containing it. A list of such objects (chained through *sod_next*) is pointed at by the *sdt_sods* in the section_dispatch_table structure.

```
struct sod {
        long    sod_name;
        u_int   sod_library : 1,
                sod_reserved : 31;
        short   sod_major;
        short   sod_minor;
        long    sod_next;
};
```

sod_name The offset in the text segment of a string describing this link object.

sod_library If set, *sod_name* specifies a library that is to be searched for by **ld.so**. The path name is obtained by searching a set of directories (see also ldconfig(8)) for a shared object matching *lib<sod_name>.so.n.m*. If not set, *sod_name* should point at a full path name for the desired shared object.

sod_major Specifies the major version number of the shared object to load.

sod_minor Specifies the preferred minor version number of the shared object to load.

The run-time link-editor maintains a list of structures called *link maps* to keep track of all shared objects loaded into a process' address space. These structures are only used at run-time and do not occur within the text or data segment of an executable or shared library.

```
struct so_map {
        caddr_t som_addr;
        char    *som_path;
        struct  so_map *som_next;
        struct  sod *som_sod;
        caddr_t som_sodbase;
        u_int   som_write : 1;
```

```
        struct  _dynamic *som_dynamic;
        caddr_t som_spd;
};
```

som_addr The address at which the shared object associated with this link map has been loaded.

som_path The full path name of the loaded object.

som_next Pointer to the next link map.

som_sod The *sod* structure that was responsible for loading this shared object.

som_sodbase Tossed out in later versions of the run-time linker.

som_write Set if (some portion of) this object's text segment is currently writable.

som_dynamic Pointer to this object's *_dynamic* structure.

som_spd Hook for attaching private data maintained by the run-time link-editor.

Symbol description with size. This is simply an *nlist* structure with one field (*nz_size*) added. Used to convey size information on items in the data segment of shared objects. An array of these lives in the shared object's text segment and is addressed by the *sdt_nzlist* field of *section_dispatch_table*.

```
struct nzlist {
        struct nlist        nlist;
        u_long              nz_size;
#define nz_un               nlist.n_un
#define nz_strx             nlist.n_un.n_strx
#define nz_name             nlist.n_un.n_name
#define nz_type             nlist.n_type
#define nz_value    nlist.n_value
#define nz_desc             nlist.n_desc
#define nz_other    nlist.n_other
};
```

nlist (see nlist(3)).

nz_size The size of the data represented by this symbol.

A hash table is included within the text segment of shared object to facilitate quick lookup of symbols during run-time link-editing. The *sdt_hash* field of the *section_dispatch_table* structure points at an array of *rrs_hash* structures:

```
struct rrs_hash {
        int     rh_symbolnum;           /* symbol number */
        int     rh_next;        /* next hash entry */
};
```

rh_symbolnum The index of the symbol in the shared object's symbol table (as given by the *ld_symbols* field).

rh_next In case of collisions, this field is the offset of the next entry in this hash table bucket. It is zero for the last bucket element.

The *rt_symbol* structure is used to keep track of run-time allocated commons and data items copied from shared objects. These items are kept on linked list and is exported through the *dd_cc* field in the *so_debug* structure (see below) for use by debuggers.

```
struct rt_symbol {
        struct nzlist           *rt_sp;
        struct rt_symbol    *rt_next;
        struct rt_symbol    *rt_link;
        caddr_t                 rt_srcaddr;
        struct so_map           *rt_smp;
};
```

rt_sp The symbol description.

rt_next Virtual address of next rt_symbol.

rt_link Next in hash bucket. Used internally by **ld.so**.

rt_srcaddr Location of the source of initialized data within a shared object.

rt_smp The shared object which is the original source of the data that this run-time symbol describes.

The *so_debug* structure is used by debuggers to gain knowledge of any shared objects that have been loaded in the process's address space as a result of run-time link-editing. Since the run-time link-editor runs as a part of process initialization, a debugger that wishes to access symbols from shared objects can only do so after the link-editor has been called from crt0. A dynamically linked binary contains a

so_debug structure which can be located by means of the *d_debug* field in *_dynamic*.

```
struct	so_debug {
	int	dd_version;
	int	dd_in_debugger;
	int	dd_sym_loaded;
	char	*dd_bpt_addr;
	int	dd_bpt_shadow;
	struct rt_symbol *dd_cc;
};
```

dd_version Version number of this interface.

dd_in_debugger Set by the debugger to indicate to the run-time linker that the program is run under control of a debugger.

dd_sym_loaded Set by the run-time linker whenever it adds symbols by loading shared objects.

dd_bpt_addr The address where a breakpoint will be set by the run-time linker to divert control to the debugger. This address is determined by the start-up module, *crt0.o*, to be some convenient place before the call to _main.

dd_bpt_shadow Contains the original instruction that was at *dd_bpt_addr*. The debugger is expected to put this instruction back before continuing the program.

dd_cc A pointer to the linked list of run-time allocated symbols that the debugger may be interested in.

The *ld_entry* structure defines a set of service routines within **ld.so**.

```
struct ld_entry {
	void	*(*dlopen)(char *, int);
	int	(*dlclose)(void *);
	void	*(*dlsym)(void *, char *);
	char	*(*dlerror)(void);
};
```

The *crt_ldso* structure defines the interface between the start-up code in crt0 and **ld.so**.

```
struct crt_ldso {
```

```
        int             crt_ba;
        int             crt_dzfd;
        int             crt_ldfd;
        struct _dynamic *crt_dp;
        char            **crt_ep;
        caddr_t         crt_bp;
        char            *crt_prog;
        char            *crt_ldso;
        struct ld_entry *crt_ldentry;
};
#define CRT_VERSION_SUN             1
#define CRT_VERSION_BSD_2       2
#define CRT_VERSION_BSD_3       3
#define CRT_VERSION_BSD_4       4
```

crt_ba The virtual address at which **ld.so** was loaded by crt0.

crt_dzfd On SunOS systems, this field contains an open file descriptor to "*/dev/zero*" used to get demand paged zeroed pages. On FreeBSD systems it contains -1.

crt_ldfd Contains an open file descriptor that was used by crt0 to load **ld.so**.

crt_dp A pointer to main's *_dynamic* structure.

crt_ep A pointer to the environment strings.

crt_bp The address at which a breakpoint will be placed by the run-time linker if the main program is run by a debugger. See *so_debug*

crt_prog
The name of the main program as determined by crt0 (CRT_VERSION_BSD3 only).

crt_ldso The path of the run-time linker as mapped by crt0 (CRT_VERSION_BSD4 only).

The *hints_header* and *hints_bucket* structures define the layout of the library hints, normally found in "*/var/run/ld.so.hints*", which is used by **ld.so** to quickly locate the shared object images in the file system. The organization of the hints file is not unlike that of an "a.out" object file, in that it contains a header determining the offset and size of a table of fixed sized hash buckets and a common string pool.

```
struct hints_header {
```

```
        long            hh_magic;
#define HH_MAGIC        011421044151
        long            hh_version;
#define LD_HINTS_VERSION_1      1
        long            hh_hashtab;
        long            hh_nbucket;
        long            hh_strtab;
        long            hh_strtab_sz;
        long            hh_ehints;
};
```

hh_magic Hints file magic number.

hh_version Interface version number.

hh_hashtab Offset of hash table.

hh_strtab Offset of string table.

hh_strtab_sz Size of strings.

hh_ehints Maximum usable offset in hints file.

```
/*
 * Hash table element in hints file.
 */
struct hints_bucket {
        int             hi_namex;
        int             hi_pathx;
        int             hi_dewey[MAXDEWEY];
        int             hi_ndewey;
#define hi_major hi_dewey[0]
#define hi_minor hi_dewey[1]
        int             hi_next;
};
```

hi_namex Index of the string identifying the library.

hi_pathx Index of the string representing the full path name of the library.

hi_dewey The version numbers of the shared library.

hi_ndewey The number of valid entries in *hi_dewey*.

hi_next Next bucket in case of hashing collisions.

CAVEATS

Only the (GNU) C compiler currently supports the creation of shared libraries. Other programming languages cannot be used.