`_DYNAMIC`
: The address of the `.dynamic` information section.

`_END_`
: The same as `_end`. The symbol has local scope and, together with the `_START_` symbol, provides a simple means of establishing an object's address range.

`_GLOBAL_OFFSET_TABLE_`
: The position-independent reference to a link-editor supplied table of addresses, the `.got` section. This table is constructed from position-independent data references that occur in objects that have been compiled with the `-K pic` option. See "Position-Independent Code" on page 143.

`_PROCEDURE_LINKAGE_TABLE_`
: The position-independent reference to a link-editor supplied table of addresses, the `.plt` section. This table is constructed from position-independent function references that occur in objects that have been compiled with the `-K pic` option. See "Position-Independent Code" on page 143.

`_START_`
: The first location within the text segment. The symbol has local scope and, together with the `_END_` symbol, provides a simple means of establishing an object's address range.

When generating an executable, the link-editor looks for additional symbols to define the executable's entry point. If a symbol was specified using the link-editor's `-e` option, that symbol is used. Otherwise the link-editor looks for the reserved symbol names `_start`, and then `main`.

Identifying Capability Requirements

Capabilities identify the attributes of a system that are required to allow code to execute. The following capabilities, in their order of precedence, are available.

- A *platform* capability - identifies a specific platform by name.
- A *machine* capability - identifies a specific machine hardware by name.
- *Hardware* capabilities - identify instruction set extensions and other hardware details with capabilities flags.
- *Software* capabilities - reflect attributes of the software environment with capabilities flags.

Each of these capabilities can be defined individually, or combined to produce a capabilities group.

Code that can only be executed when certain capabilities are available should identify these requirements by means of a capabilities section within the associated ELF object. Recording capability requirements within an object allows the system to validate the object before attempting to execute the associated code. These requirements can also provide a framework

where the system can select the most appropriate object from a family of objects. A family consists of variants of the same object, where each variant requires different capabilities.

Dynamic objects, as well as individual functions within an object, can be associated with capability requirements. Ideally, capability requirements are recorded in the relocatable objects that are produced by the compiler, and reflect the options or optimization that was specified at compile time. The link-editor combines the capabilities of any input relocatable objects to create a final capabilities section for the output file. See "Capabilities Section" on page 243.

In addition, capabilities can be defined when the link-editor creates an output file. These capabilities are identified using a `mapfile` and the link-editor's `-M` option. Capabilities that are defined by using a `mapfile` can augment, or override, the capabilities that are specified within any input relocatable objects. `Mapfiles` are usually used to augment compilers that do not generate the necessary capability information.

System capabilities are the capabilities that describe a running system. The platform name, and machine hardware name can be displayed with `uname(1)` using the `-i` option and `-m` option respectively. The system hardware capabilities can be displayed with `isainfo(1)` using the `-v` option. At runtime, the capability requirements of an object are compared to the system capabilities to determine whether the object can be loaded, or a function within the object can be used.

Object capabilities are capabilities that are associated with an object. These capabilities define the requirements of the entire object, and control whether the object can be loaded at runtime. If an object requires capabilities that can not be satisfied by the system, then the object can not be loaded at runtime. Capabilities can be used to provide more than one instance of a given object, each optimized for systems that match the objects requirements. The runtime linker transparently selects the best instance from such a family of object instances by comparing the objects capability requirements to the capabilities provided by the system.

Symbol capabilities are capabilities that are associated with individual functions within an object. These capabilities define the requirements of one or more functions within an object, and control whether the function can be used at runtime. These capabilities allow for the presence of multiple instances of a function within a single object. Each instance of the function can be optimized for a system with different capabilities. If a function instance requires capabilities that can not be satisfied by the system, then that function instance can not be used at runtime. Instead, an alternative instance of the same function must be used. Symbol capabilities offer the ability to construct a single object that can be used on systems of varying abilities. A family of functions can provide optimized instances for systems that can support the capabilities, and more generic instances for other, less capable systems. The runtime linker transparently selects the best instance from such a family of function instances by comparing the functions capability requirements to the capabilities provided by the system.

Object and symbol capabilities provide for selecting the best object, and the best function within an object, for the currently running system. Object and symbol capabilities are optional features, both independent of each other. However, an object that defines symbol capabilities

may also define object capabilities. In this case, any family of capabilities functions should be accompanied with one instance of the function that satisfies the object capabilities. If no object capabilities exist, any family of capability functions should be accompanied with one instance of the function that requires no capabilities. This function instance provides the default implementation, should no capability instance be applicable for a given system.

The following x86 example displays the object capabilities of `foo.o`. These capabilities apply to the entire object. This object contains no symbol capabilities functions.

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
```

The following x86 example displays the symbol capabilities of `bar.o`. These capabilities apply to the individual functions `foo` and `bar`. Two instances of each symbol exist, each instance being assigned to a different set of capabilities. In this example, no object capabilities exist.

```
$ elfdump -H bar.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [25]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
      [26]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
     index  tag               value
       [3]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [33]  0x00000044 0x00000021  FUNC LOCL  D    0 .text    foo%sse
      [34]  0x00000068 0x0000001e  FUNC LOCL  D    0 .text    bar%sse
```

Note – In this example, the capability symbols follow a naming convention that appends a capability identifier to the generic symbol name. This convention can be produced by the link-editor when object capabilities are converted to symbol capabilities, and is discussed later in "Converting Object Capabilities to Symbol Capabilities" on page 77.

Capability definitions provide for many combinations that allow you to identify the requirements of an object, or functions within an object. Hardware capabilities provide the greatest flexibility. Hardware capabilities define hardware requirements without dictating a specific machine hardware name, or platform name. However, sometimes there are attributes of an underlying system that can only be determined from the machine hardware name, or platform name. Identifying a capability name can allow you to code to very specific system capabilities, but the use of the identified object can be restrictive. Should a new machine hardware name or platform name become applicable for the object, the object must be rebuilt to identify the new capability name.

The following sections describe how capabilities can be defined, and used by the link-editors.

Identifying a Platform Capability

A platform capability of an object identifies the platform name that the object, or specific function, can execute upon. Multiple platform capabilities can be defined. This identification is very specific, and takes precedence over any other capability types.

The platform name of a system can be displayed by the utility `uname(1)` with the `-i` option.

A platform capability requirement can be defined using the following `mapfile` syntax.

```
platcap = name,... [ OVERRIDE ];
```

The `platcap` declaration is qualified with one or more platform names. The following SPARC example identifies the object `foo.so.1` as being specific to the `SUNW,SPARC-Enterprise` platform.

```
$ cat mapfile
platcap = SUNW,SPARC-Enterprise;
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_PLAT     SUNW,SPARC-Enterprise
```

Relocatable objects can define platform capabilities. These capabilities are gathered together to define the final capability requirements of the object being built.

The platform capability of an object being built can be controlled explicitly from a `mapfile` by using the `OVERRIDE` keyword. An `OVERRIDE` keyword, together with a platform capability name, overrides any platform capabilities that might be provided from any input relocatable objects. An `OVERRIDE` keyword, together with a null platform capability name, or a value of 0, effectively removes any platform capability requirement from the object being built.

A platform capability requirement defined in a dynamic object is validated by the runtime linker against the platform name of the system. The object is only used if one of the platform names recorded in the object match the platform name of the system.

Targeting code to a specific platform can be useful in some instances, however the development of a hardware capabilities family can provide greater flexibility, and is recommended. Hardware capabilities families can provide for optimized code to be exercised on a broader range of systems.

Identifying a Machine Capability

A machine capability of an object identifies the machine hardware name that the object, or specific function, can execute upon. Multiple machine capabilities can be defined. This identification carries less precedence than platform capability definitions, but takes precedence over any other capability types.

The machine hardware name of a system can be displayed by the utility `uname(1)` with the `-m` option.

A machine capability requirement can be defined using the following `mapfile` syntax.

```
machcap = name,... [ OVERRIDE ];
```

The `machcap` declaration is qualified with one or more machine hardware names. The following SPARC example identifies the object `foo.so.1` as being specific to the `sun4u` machine hardware name.

```
$ cat mapfile
machcap = sun4u;
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_MACH     sun4u
```

Relocatable objects can define machine capabilities. These capabilities are gathered together to define the final capability requirements of the object being built.

The machine capability of an object being built can be controlled explicitly from a `mapfile` by using the `OVERRIDE` keyword. An `OVERRIDE` keyword, together with a machine capability name, overrides any machine capabilities that might be provided from any input relocatable objects. An `OVERRIDE` keyword, together with a null machine capability name, or a value of 0, effectively removes any machine capability requirement from the object being built.

A machine capability requirement defined in a dynamic object is validated by the runtime linker against the machine hardware name of the system. The object is only used if one of the machine names recorded in the object match the machine name of the system.

Targeting code to a specific machine can be useful in some instances, however the development of a hardware capabilities family can provide greater flexibility, and is recommended. Hardware capabilities families can provide for optimized code to be exercised on a broader range of systems.

Identifying Hardware Capabilities

The hardware capabilities of an object identify the hardware requirements of a system necessary for the object, or specific function, to execute correctly. An example of this requirement might be the identification of code that requires the `MMX` or `SSE` features that are available on some x86 architectures.

Hardware capability requirements can be identified using the following `mapfile` syntax.

```
hwcap_1 = TOKEN | Vval [ OVERRIDE ];
```

The `hwcap_1` declaration is qualified with one or more tokens, which are symbolic representations of hardware capabilities. In addition, or alternatively, a numeric value representing one of more capabilities can be supplied by prefixing the value with a `V`.

For SPARC systems, hardware capabilities are defined as `AV_` values in `sys/auxv_SPARC.h`. For x86 systems, hardware capabilities are defined as `AV_` values in `sys/auxv_386.h`.

The following x86 example shows the declaration of `MMX` and `SSE` as hardware capabilities required by the object `foo.so.1`.

```
$ egrep "MMX|SSE" /usr/include/sys/auxv_386.h
#define AV_386_MMX    0x0040
#define AV_386_SSE    0x0800
$ cat mapfile
hwcap_1 = SSE MMX;
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap
 Object Capabilities:
     index  tag               value
      [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
```

Relocatable objects can contain hardware capabilities values. The link-editor combines any hardware capabilities values from multiple input relocatable objects. The resulting `CA_SUNW_HW_1` value is a bitwise-inclusive OR of the associated input values. By default, these values are combined with the hardware capabilities specified by a `mapfile`.

The hardware capability requirements of an object can be controlled explicitly from a `mapfile` by using the `OVERRIDE` keyword. An `OVERRIDE` keyword, together with a hardware capability, overrides any hardware capabilities that might be provided from any input relocatable objects. An `OVERRIDE` keyword, together with a hardware capability value of 0, effectively removes any hardware capabilities requirement from the object being built.

The following example suppresses any hardware capabilities data defined by the input relocatable object `foo.o` from being included in the output file, `bar.o`.

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap
 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
$ cat mapfile
hwcap_1 = V0x0 OVERRIDE;
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

Any hardware capability requirements defined by a dynamic object are validated by the runtime linker against the hardware capabilities that are provided by the system. If any of the hardware capability requirements can not be satisfied, the object is not loaded at runtime. For example, if the `SSE` feature is not available to a process, `ldd(1)` indicates the following error.

```
$ ldd prog
        foo.so.1 =>      ./foo.so.1  - hardware capability unsupported: \
                         0x800 [ SSE ]
        ....
```

Multiple variants of a dynamic object that exploit different hardware capabilities can provide a flexible runtime environment using filters. See "Capability Specific Shared Objects" on page 379.

Hardware capabilities can also be used to identify the capabilities of individual functions within a single object. In this case, the runtime linker can select the most appropriate function instance to use based upon the current system capabilities. See "Creating a Symbol Capabilities Family" on page 73.

Identifying Software Capabilities

The software capabilities of an object identify characteristics of the software that might be important for debugging or monitoring processes. Software capabilities can also influence process execution. Presently, the only software capabilities that are recognized relate to frame pointer usage by the object, and process address space restrictions.

Objects can indicate that their frame pointer use is known. This state is then qualified by declaring the frame pointer as being used or not.

64–bit objects can indicate that at runtime they must be exercised within a 32–bit address space.

Software capabilities flags are defined in `sys/elf.h`.

```
#define  SF1_SUNW_FPKNWN    0x001
#define  SF1_SUNW_FPUSED    0x002
#define  SF1_SUNW_ADDR32    0x004
```

These software capability requirements can be identified using the following `mapfile` syntax.

```
sfcap_1 = TOKEN | Vval [ OVERRIDE ];
```

The `sfcap_1` declaration can be qualified with the tokens `FPKNWN`, `FPUSED` and `ADDR32`. Or, alternatively with a numeric value that represents these states.

Relocatable objects can contain software capabilities values. The link-editor combines the software capabilities values from multiple input relocatable objects. Software capabilities can also be supplied with a `mapfile`. By default, any `mapfile` values are combined with the values supplied by relocatable objects.

The software capability requirements of an object can be controlled explicitly from a `mapfile` by using the `OVERRIDE` keyword. An `OVERRIDE` keyword, together with a software capability value of 0, effectively removes any software capabilities requirement from the object being built.

The following example suppresses any software capabilities data defined by the input relocatable object `foo.o` from being included in the output file, `bar.o`.

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap
 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_SF_1     0x3  [ SF1_SUNW_FPKNWN  SF1_SUNW_FPUSED ]
$ cat mapfile
sfcap_1 = V0x0 OVERRIDE;
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

Software Capability Frame Pointer Processing

The computation of a CA_SUNW_SF_1 value from two frame pointer input values is as follows.

TABLE 2–1 CA_SUNW_SF_1 Frame Pointer Flag Combination State Table

Input file 1	Input file 2		
	SF1_SUNW_FPKNWN **SF1_SUNW_FPUSED**	**SF1_SUNW_FPKNWN**	**<unknown>**
SF1_SUNW_FPKNWN **SF1_SUNW_FPUSED**	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED
SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN
<unknown>	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED	SF1_SUNW_FPKNWN	<unknown>

This computation is applied to each relocatable object value and mapfile value. The frame pointer software capabilities of an object are unknown if no .SUNW_cap section exists, or if the section contains no CA_SUNW_SF_1 value, or if neither the SF1_SUNW_FPKNW or SF1_SUNW_FPUSED flags are set.

Software Capability Address Space Restriction Processing

64–bit objects that are identified with the SF1_SUNW_ADDR32 software capabilities flag can contain optimized code that requires a 32–bit address space. 64–bit objects that are identified in this manner can interoperate with any other 64–bit objects whether they are identified with the SF1_SUNW_ADDR32 flag or not. An occurrence of the SF1_SUNW_ADDR32 flag within a 64–bit input relocatable object is propagated to the CA_SUNW_SF_1 value that is created for the output file being created by the link-editor.

The existence of the SF1_SUNW_ADDR32 flag within a 64–bit executable ensures that the associated process is restricted to the lower 32–bit address space. This restricted address space includes the process stack and all process dependencies. Within such a process, all objects, whether they are identified with the SF1_SUNW_ADDR32 flag or not, are loaded within the restricted 32–bit address space.

64–bit shared objects can contain the SF1_SUNW_ADDR32 flag. However, the restricted address space requirement can only be established by a 64–bit executable containing the SF1_SUNW_ADDR32 flag. Therefore, a 64–bit SF1_SUNW_ADDR32 shared object must be a dependency of a 64–bit SF1_SUNW_ADDR32 executable.

A 64–bit SF1_SUNW_ADDR32 shared object that is encountered by the link-editor when building an unrestricted 64–bit executable results in a warning.

```
$ cc -m64 -o main main.c -lfoo
ld: warning: file libfoo.so: section .SUNW_cap: software capability ADDR32: \
    requires executable be built with ADDR32 capability
```

A 64–bit SF1_SUNW_ADDR32 shared object that is encountered at runtime by a process that is created from an unrestricted 64–bit executable, results in a fatal error.

```
$ ldd main
        libfoo.so =>     ./libfoo.so  - software capability unsupported: \
                         0x4  [ ADDR32 ]
        ....
$ main
ld.so.1: main: fatal: ./libfoo.so: software capability unsupported: 0x4  [ ADDR32 ]
```

An executable can be seeded with the SF1_SUNW_ADDR32 using a mapfile.

```
$ cat mapfile
sfcap_1 = ADDR32;
$ cc -m64 -o main main.c -Mmapfile -lfoo
$ elfdump -H main

Capabilities Section:  .SUNW_cap
 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_SF_1     0x4  [ SF1_SUNW_ADDR32 ]
```

Creating a Symbol Capabilities Family

Developers often desire to provide multiple instances of functions, each optimized for a particular set of capabilities, within a single object. It is desirable for the selection and use of these instances to be transparent to any consumers. A generic, front-end function can be created to provide an external interface. This generic instance, together with the optimized instances, can be combined into one object. The generic instance might use getisax(2) to determine the systems capabilities and then call the appropriate optimized function instance to handle a task. Although this model works, it suffers from a lack of generality, and incurs a runtime overhead.

Symbol capabilities offer an alternative mechanism to construct such an object. This mechanism is simpler, more efficient, and does not require you to write additional front-end code. Multiple instances of a function can be created and associated with different capabilities. These instances, together with a default instance of the function that is suitable for any system, can be combined into a single dynamic object. The selection of the most appropriate member from this family of symbols is carried out by the runtime linker using the symbol capabilities information.

In the following example, the x86 objects foobar.mmx.o and foobar.sse.o, contain the same function foo() and bar(), that have been compiled to use the MMX and SSE instructions respectively.

```
$ elfdump -H foobar.mmx.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        mmx
       [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [10]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
      [16]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx

$ elfdump -H foobar.sse.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        sse
       [2]  CA_SUNW_HW_1      0x800  [ SSE ]

  Capabilities symbols:
     index    value      size      type bind oth ver shndx    name
      [16]  0x00000000 0x0000002f  FUNC LOCL  D    0 .text    foo%sse
      [18]  0x00000048 0x00000030  FUNC LOCL  D    0 .text    bar%sse
```

Each of these objects contain a local symbol identifying the capabilities function `foo%*()` and `bar%*()`. In addition, each object also defines a global reference to the function `foo()` and `bar()`. Any internal references to `foo()` or `bar()` are relocated through these global references, as are any external interfaces.

These two objects can now be combined with a default instance of `foo()` and `bar()`. These default instances satisfy the global references, and provide an implementation that is compatible with any object capabilities. These default instances are said to lead each capabilities family. If no object capabilities exist, this default instance should also require no capabilities. Effectively, three instances of `foo()` and `bar()` exist, the global instance provides the default, and the local instances provide implementations that are used at runtime if the associated capabilities are available.

```
$ cc -o libfoobar.so.1 -G foobar.o foobar.sse.o foobar.mmx.o
$ elfdump -sN.dynsym libfoobar.so.1 | egrep "foo|bar"
       [2]  0x00000700 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
       [4]  0x00000750 0x0000002f  FUNC LOCL  D    0 .text    foo%sse
       [8]  0x00000784 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx
       [9]  0x000007b0 0x00000030  FUNC LOCL  D    0 .text    bar%sse
      [15]  0x000007a0 0x00000014  FUNC GLOB  D    1 .text    foo
```

```
      [17]  0x000007c0 0x00000014  FUNC GLOB  D    1 .text    bar
```

The capabilities information for a dynamic object displays the capabilities symbols, and reveals the capabilities families that are available.

```
$ elfdump -H libfoobar.so.1

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        mmx
       [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
       [2]  0x00000700 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
       [8]  0x00000784 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
     index  tag               value
       [4]  CA_SUNW_ID        sse
       [5]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
       [4]  0x00000750 0x0000002f  FUNC LOCL  D    0 .text    foo%sse
       [9]  0x000007b0 0x00000030  FUNC LOCL  D    0 .text    bar%sse

Capabilities Chain Section:  .SUNW_capchain

 Capabilities family: foo
  chainndx  symndx      name
         1  [15]        foo
         2  [2]         foo%mmx
         3  [4]         foo%sse

 Capabilities family: bar
  chainndx  symndx      name
         5  [17]        bar
         6  [8]         bar%mmx
         7  [9]         bar%sse
```

At runtime, all references to `foo()` and `bar()` are initially bound to the global symbols. However, the runtime linker recognizes that these functions are the lead instance of a capabilities family. The runtime linker inspects each family member to determine if a better capability function is available. There is a one time cost to this operation, which occurs on the

first call to the function. Subsequent calls to `foo()` and `bar()` are bound directly to the function instance selected by the first call. This function selection can be observed by using the runtime linkers debugging capabilities.

In the following example, the underlying system does not provide `MMX` or `SSE` support. The lead instance of `foo()` requires no special capabilities support, and thus satisfies any relocation reference.

```
$ LD_DEBUG=symbols main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo%mmx[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo%mmx[2]:  capability rejected
debug:   symbol=foo%sse[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo%sse[4]:  capability rejected
debug:   symbol=foo[15]:  used
```

In the following example, `MMX` is available, but `SSE` is not. The `MMX` capable instance of `foo()` satisfies any relocation reference.

```
$ LD_DEBUG=symbols main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo[2]:  capability candidate
debug:   symbol=foo[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo[4]:  capability rejected
debug:   symbol=foo[2]:  used
```

A family of capabilities function instances must be accessed from a procedure linkage table entry. See "Procedure Linkage Table (Processor-Specific)" on page 313. This procedure linkage reference requires the runtime linker to resolve the function. During this process, the runtime linker can process the associated symbol capabilities information, and select the best function from the available family of function instances.

When symbol capabilities are not used, there are cases where the link-editor can resolve references to code without the need of a procedure linkage table entry. For example, within a dynamic executable, a reference to a function that exists within the executable can be bound internally at link-edit time. Hidden and protected functions within shared objects can also be bound internally at link-edit time. In these cases, there is normally no need for the runtime linker to be involved in resolving a reference to these functions.

However, when symbol capabilities are used, the function must be resolved from a procedure linkage table entry. This entry is necessary in order for the runtime linker to be involved in selecting the appropriate function, while maintaining a read-only text segment. This

mechanism results in an indirection through a procedure linkage table entry for all calls to a capability function. This indirection might not be necessary if symbol capabilities are not used. Therefore, there is a small trade off between the cost of calling the capability function, and any performance improvement gained from using the capability function over its default counterpart.

Note – Although a capability function must be accessed through a procedure linkage table entry, the function can still be defined as hidden or protected. The runtime linker honors these visibility states and restricts any binding to these functions. This behavior results in the same bindings as are produced when symbol capabilities are not associated with the function. A hidden function can not be bound to from an external object. A reference to a protected function from within an object will only be bound to within the same object.

Converting Object Capabilities to Symbol Capabilities

Ideally, the compilation environment can generate objects that are identified with symbol capabilities. If the compilation environment can not create symbol capabilities, the link-editor offers a solution.

A relocatable object that defines object capabilities can be transformed into a relocatable object that defines symbol capabilities using the link-editor. Using the link-editor `-z symbolcap` option, any object capability data is converted to symbol capability data. All global functions within the object are converted into local functions, and are associated to the symbol capabilities. These transformed symbols are appended with any capability identifier specified as part of the object capabilities group. If a capability identifier is not defined, a default group name is appended.

For each original global function, a global reference is created. This reference is associated to any relocation requirements, and provides for binding to a default, global function when this object is finally combined to create a dynamic executable or shared object.

Note – The `-z symbolcap` option applies to objects that contain an object capabilities section. The option has no affect upon relocatable objects that already contain symbol capabilities, or relocatable objects that contain both object and symbol capabilities. This design allows multiple objects to be combined by the link-editor, with only those objects that contain object capabilities being affected by the option.

In the following example, a x86 relocatable object contains two global functions `foo()` and `bar()`. This object has been compiled to require the `MMX` and `SSE` hardware capabilities. In these examples, the capabilities group has been named with a capabilities identifier entry. This identifier name is appended to the transformed symbol names. Without this explicit identifier, the link-editor appends a default capabilities group name.

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_ID        sse,mmx
       [1]  CA_SUNW_HW_1      0x840  [ SSE MMX ]

$ elfdump -s foo.o | egrep "foo|bar"
      [25]  0x00000000 0x00000021  FUNC GLOB  D    0 .text    foo
      [26]  0x00000024 0x0000001e  FUNC GLOB  D    0 .text    bar

$ elfdump -r foo.o | fgrep foo
  R_386_PLT32               0x38       .rel.text             foo
```

This relocatable object can now be transformed into a symbols capabilities relocatable object.

```
$ ld -r -o foo.1.o -z symbolcap foo.o
$ elfdump -H foo.1.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        sse,mmx
       [2]  CA_SUNW_HW_1      0x840  [ SSE MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [25]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%sse,mmx
      [26]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%sse,mmx

$ elfdump -s foo.1.o | egrep "foo|bar"
      [25]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%sse,mmx
      [26]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%sse,mmx
      [37]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF    foo
      [38]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF    bar

$ elfdump -r foo.1.o | fgrep foo
  R_386_PLT32               0x38       .rel.text             foo
```

This object can now be combined with other objects containing instances of the same functions, associated with different symbol capabilities, to produce an executable or shared object. In addition, a default instance of each function, one that is not associated with any symbol capabilities, should be provided to lead each capabilities family. This default instance provides for all external references, and ensures that an instance of the function is available on any system.

At runtime, any references to `foo()` and `bar()` are directed to the lead instances. However, the runtime linker selects the best symbol capabilities instance if the system accommodates the appropriate capabilities.

Exercising a Capability Family

Objects are normally designed and built so that they can execute on all systems of a given architecture. However, individual systems, with special capabilities, are often targeted for optimization. Optimized code can be identified with the capabilities that the code requires to execute, using the mechanisms described in the previous sections.

To exercise and test optimized instances it is necessary to use a system that provides the required capabilities. For each system, the runtime linker determines the capabilities that are available, and then chooses the most capable instances. To aid testing and experimentation, the runtime linker can be told to use an alternative set of capabilities than those provided by the system. In addition, you can specify that only specific files should be validated against these alternative capabilities.

An alternative set of capabilities are derived from the system capabilities, and can be re-initialized or have capabilities added or removed.

A family of environment variables are available to create and target the use of an alternative set of capabilities.

`LD_PLATCAP={`*name*`}`	Identifies an alternative platform name.
`LD_MACHCAP={`*name*`}`	Identifies an alternative machine hardware name.
`LD_HWCAP=[+-]{`*token* `|` *number*`},...`	Identifies an alternative hardware capabilities value.
`LD_SFCAP=[+-]{`*token* `|` *number*`},...`	Identifies an alternative software capabilities value.
`LD_CAP_FILES=`*file*`,...`	Identifies the files that should be validated against the alternative capabilities.

The capabilities environment variables `LD_PLATCAP` and `LD_MACHCAP` accept a string that defines the platform name and machine hardware names respectively. See "Identifying a Platform Capability" on page 67, and "Identifying a Machine Capability" on page 68.

The capabilities environment variables `LD_HWCAP` and `LD_SFCAP` accept a comma separated list of *tokens* as a symbolic representation of capabilities. See "Identifying Hardware Capabilities" on page 69, and "Identifying Software Capabilities" on page 71. A token can also be a numeric value.

A “+” prefix results in the capabilities that follow being added to the alternative capabilities. A “-” prefix results in the capabilities that follow being removed from the alternative capabilities. The lack of “+-” result in the capabilities that follow replacing the alternative capabilities.

The removal of a capability results in a more restricted capabilities environment being emulated. Normally, when a family of capabilities instances are available, a generic, non-capabilities specific instance is also provided. A more restricted capabilities environment can therefore be used to force the use of less capable, or generic code instances.

The addition of a capability results in a more enhanced capabilities environment being emulated. This environment should be created with caution, but can be used to exercise the framework of a capabilities family. For example, a family of functions can be created that define their expected capabilities using mapfiles. These functions can use `printf(3C)` to confirm their execution. The creation of the associated objects can then be validated and exercised with various capability combinations. This prototyping of a capabilities family can prove useful before the real capabilities requirements of the functions are coded. However, if the code within a family instance requires a specific capability to execute correctly, and this capability is not provided by the system, but is set as an alternative capability, the code instance will fail to execute correctly.

Establishing a set of alternative capabilities without also using `LD_CAP_FILES` results in all of the capabilities specific objects of a process being validated against the alternative capabilities. This approach should also be exercised with caution, as many system objects require system capabilities to execute correctly. Any alteration of capabilities can cause system objects to fail to execute correctly.

A best environment for capabilities experimentation is to use a system that provides all the capabilities your objects are targeted to use. `LD_CAP_FILES` should also be used to isolate the objects you wish to experiment with. Capabilities can then be disabled, using the “-” syntax, so that the various instances of your capabilities family can be exercised. Each instance is fully supported by the true capabilities of the system.

For example, suppose you have two x86 capabilities objects, `libfoo.so` and `libbar.so`. These objects contain capability functions optimized to use `SSE2` instructions, functions optimized to use `MMX` instructions, and generic functions that require no capabilities. The underlying system provides both `SSE2` and `MMX`. By default, the fully optimized `SSE2` functions are used.

`libfoo.so` and `libbar.so` can be restricted to use the functions optimized for `MMX` instructions by removing the `SSE2` capability by using a `LD_HWCAP` definition. The most flexible means of defining `LD_CAP_FILES` is to use the base name of the required files.

```
$ LD_HWCAP=-sse2 LD_CAP_FILES=libfoo.so,libbar.so ./main
```

`libfoo.so` and `libbar.so` can be further restricted to use only generic functions by removing the `SSE2` and `MMX` capabilities.

```
$ LD_HWCAP=-sse2,mmx LD_CAP_FILES=libfoo.so,libbar.so ./main
```

Note – The capabilities available for an application, and any alternative capabilities that have been set, can be observed using the runtime linkers diagnostics.

```
$ LD_DEBUG=basic LD_HWCAP=-sse2,mmx,cx8 ./main
....
02328: hardware capabilities (CA_SUNW_HW_1) - 0x5c6f  \
    [ SSE3 SSE2 SSE FXSR MMX CMOV SEP CX8 TSC FPU ]
02328: alternative hardware capabilities (CA_SUNW_HW_1) - 0x4c2b  \
    [ SSE3 SSE FXSR CMOV SEP TSC FPU ]
....
```

Relocation Processing

After you have created the output file, all data sections from the input files are copied to the new image. Any relocations specified by the input files are applied to the output image. Any additional relocation information that must be generated is also written to the new image.

Relocation processing is normally uneventful, although error conditions might arise that are accompanied by specific error messages. Two conditions are worth more discussion. The first condition involves text relocations that result from position-dependent code. This condition is covered in more detail in "Position-Independent Code" on page 143. The second condition can arise from displacement relocations, which is described more fully in the next section.

Displacement Relocations

Error conditions might occur if displacement relocations are applied to a data item, which can be used in a copy relocation. The details of copy relocations are covered in "Copy Relocations" on page 150.

A displacement relocation remains valid when both the relocated offset and the relocation target remain separated by the same displacement. A copy relocation is where a global data item within a shared object is copied to the `.bss` of an executable. This copy preserves the executable's read-only text segment. If the copied data has a displacement relocation applied to the data, or an external relocation is a displacement into the copied data, the displacement relocation becomes invalidated.

Two areas of validation attempt to catch displacement relocation problems.

TABLE 7–5 ELF Section Types, *sh_type* *(Continued)*

Name	Value
SHT_GROUP	17
SHT_SYMTAB_SHNDX	18
SHT_LOOS	0x60000000
SHT_LOSUNW	0x6fffef
SHT_SUNW_capchain	0x6ffffef
SHT_SUNW_capinfo	0x6ffffff0
SHT_SUNW_symsort	0x6ffffff1
SHT_SUNW_tlssort	0x6ffffff2
SHT_SUNW_LDYNSYM	0x6ffffff3
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_cap	0x6ffffff5
SHT_SUNW_SIGNATURE	0x6ffffff6
SHT_SUNW_ANNOTATE	0x6ffffff7
SHT_SUNW_DEBUGSTR	0x6ffffff8
SHT_SUNW_DEBUG	0x6ffffff9
SHT_SUNW_move	0x6ffffffa
SHT_SUNW_COMDAT	0x6ffffffb
SHT_SUNW_syminfo	0x6ffffffc
SHT_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
SHT_HISUNW	0x6fffffff
SHT_HIOS	0x6fffffff
SHT_LOPROC	0x70000000
SHT_SPARC_GOTDATA	0x70000000
SHT_AMD64_UNWIND	0x70000001
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000

header indexes against which the symbol table entries are defined. Only if corresponding symbol table entry's `st_shndx` field contains the escape value `SHN_XINDEX` will the matching `Elf32_Word` hold the actual section header index. Otherwise, the entry must be `SHN_UNDEF` (`0`).

`SHT_LOOS` – `SHT_HIOS`
Values in this inclusive range are reserved for operating system-specific semantics.

`SHT_LOSUNW` – `SHT_HISUNW`
Values in this inclusive range are reserved for Solaris OS semantics.

`SHT_SUNW_capchain`
An array of indices that collect capability family members. The first element of the array is the chain version number. Following this element are a chain of `0` terminated capability symbol indices. Each `0` terminated group of indices represents a capabilities family. The first element of each family is the capabilities lead symbol. The following elements point to family members. See "Capabilities Section" on page 243 for details.

`SHT_SUNW_capinfo`
An array of indices that associate symbol table entries to capabilities requirements, and their lead capabilities symbol. An object that defines symbol capabilities contains a `SHT_SUNW_cap` section. The `SHT_SUNW_cap` section header information points to the associated `SHT_SUNW_capinfo` section. The `SHT_SUNW_capinfo` section header information points to the associated symbol table section. See "Capabilities Section" on page 243 for details.

`SHT_SUNW_symsort`
An array of indices into the dynamic symbol table that is formed by the adjacent `SHT_SUNW_LDYNSYM` section and `SHT_DYNSYM` section. These indices are relative to the start of the `SHT_SUNW_LDYNSYM` section. The indices reference those symbols that contain memory addresses. The indices are sorted such that the indices reference the symbols by increasing address.

`SHT_SUNW_tlssort`
An array of indices into the dynamic symbol table that is formed by the adjacent `SHT_SUNW_LDYNSYM` section and `SHT_DYNSYM` section. These indices are relative to the start of the `SHT_SUNW_LDYNSYM` section. The indices reference thread-local storage symbols. See Chapter 8, "Thread-Local Storage." The indices are sorted such that the indices reference the symbols by increasing offset.

`SHT_SUNW_LDYNSYM`
Dynamic symbol table for non-global symbols. See previous `SHT_SYMTAB`, `SHT_DYNSYM`, `SHT_SUNW_LDYNSYM` description.

`SHT_SUNW_dof`
Reserved for internal use by `dtrace(1M)`.

`SHT_SUNW_cap`
Specifies capability requirements. See "Capabilities Section" on page 243 for details.

TABLE 7–9 ELF sh_link and sh_info Interpretation *(Continued)*

sh_type	sh_link	sh_info
SHT_SUNW_cap	If symbol capabilities exist, the section header index of the associated SHT_SUNW_capinfo table, otherwise 0.	If any capabilities refer to named strings, the section header index of the associated string table, otherwise 0.
SHT_SUNW_capinfo	The section header index of the associated symbol table.	For a dynamic object, the section header index of the associated SHT_SUNW_capchain table, otherwise 0.
SHT_SUNW_symsort	The section header index of the associated symbol table.	0
SHT_SUNW_tlssort	The section header index of the associated symbol table.	0
SHT_SUNW_LDYNSYM	The section header index of the associated string table. This index is the same string table used by the SHT_DYNSYM section.	One greater than the symbol table index of the last local symbol, STB_LOCAL. Since SHT_SUNW_LDYNSYM only contains local symbols, sh_info is equivalent to the number of symbols in the table.
SHT_SUNW_move	The section header index of the associated symbol table.	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	The section header index of the associated symbol table.	The section header index of the associated .dynamic section.
SHT_SUNW_verdef	The section header index of the associated string table.	The number of version definitions within the section.
SHT_SUNW_verneed	The section header index of the associated string table.	The number of version dependencies within the section.
SHT_SUNW_versym	The section header index of the associated symbol table.	0

Section Merging

The SHF_MERGE section flag can be used to mark SHT_PROGBITS sections within relocatable objects. See Table 7–8. This flag indicates that the section can be merged with compatible sections from other objects. Such merging has the potential to reduce the size of any executable or shared object that is built from these relocatable objects. This size reduction can also have a positive effect on the runtime performance of the resulting object.

TABLE 7–10 ELF Special Sections *(Continued)*

Name	Type	Attribute
.shstrtab	SHT_STRTAB	None
.strtab	SHT_STRTAB	Refer to the explanation following this table.
.symtab	SHT_SYMTAB	See "Symbol Table Section" on page 265
.symtab_shndx	SHT_SYMTAB_SHNDX	See "Symbol Table Section" on page 265
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata, .tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_cap	SHT_SUNW_cap	SHF_ALLOC
.SUNW_capchain	SHT_SUNW_capchain	SHF_ALLOC
.SUNW_capinfo	SHT_SUNW_capinfo	SHF_ALLOC
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_ldynsym	SHT_SUNW_LDYNSYM	SHF_ALLOC
.SUNW_dynsymsort	SHT_SUNW_symsort	SHF_ALLOC
.SUNW_dymtlssort	SHT_SUNW_tlssort	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_REL	SHF_ALLOC
	SHT_RELA	
.SUNW_syminfo	SHT_SUNW_syminfo	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef	SHF_ALLOC
	SHT_SUNW_verneed	
	SHT_SUNW_versym	

`.bss`
: Uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type `SHT_NOBITS`.

`.comment`
: Comment information, typically contributed by the components of the compilation system. This section can be manipulated by `mcs(1)`.

`.symtab_shndx`
This section holds the special symbol table section index array, as described by `.symtab`. The section's attributes include the `SHF_ALLOC` bit if the associated symbol table section does. Otherwise, that bit is turned off.

`.tbss`
This section holds uninitialized thread-local data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the data is instantiated for each new execution flow. The section occupies no file space, as indicated by the section type, `SHT_NOBITS`. See Chapter 8, "Thread-Local Storage," for details.

`.tdata`, `.tdata1`
These sections hold initialized thread-local data that contribute to the program's memory image. A copy of its contents is instantiated by the system for each new execution flow. See Chapter 8, "Thread-Local Storage," for details.

`.text`
The *text* or executable instructions of a program.

`.SUNW_bss`
Partially initialized data for shared objects that contribute to the program's memory image. The data is initialized at runtime. The section occupies no file space, as indicated by the section type `SHT_NOBITS`.

`.SUNW_cap`
Capability requirements. See "Capabilities Section" on page 243 for details.

`.SUNW_capchain`
Capability chain table. See "Capabilities Section" on page 243 for details.

`.SUNW_capinfo`
Capability symbol information. See "Capabilities Section" on page 243 for details.

`.SUNW_heap`
The *heap* of a dynamic executable created from `dldump(3C)`.

`.SUNW_dynsymsort`
An array of indices to symbols in the combined `.SUNW_ldynsym` – `.dynsym` symbol table. The indices are sorted to reference symbols in order of increasing address. Symbols that do not represent variables or do not represent functions are not included. In the case of redundant global symbols and weak symbols, only the weak symbol is kept. See "Symbol Sort Sections" on page 273 for details.

`.SUNW_dyntlssort`
An array of indices to thread-local storage symbols in the combined `.SUNW_ldynsym` – `.dynsym` symbol table. The indices are sorted to reference symbols in order of increasing offset. Symbols that do not represent TLS variables are not included. In the case of redundant global symbols and weak symbols, only the weak symbol is kept. See "Symbol Sort Sections" on page 273 for details.

The section header indices in the SHT_GROUP section, identify the sections that make up the group. These sections must have the SHF_GROUP flag set in their sh_flags section header member. If the link-editor decides to remove the section group, the link-editor removes all members of the group.

To facilitate removing a group without leaving dangling references and with only minimal processing of the symbol table, the following rules are followed.

- References to the sections comprising a group from sections outside of the group must be made through symbol table entries with STB_GLOBAL or STB_WEAK binding and section index SHN_UNDEF. A definition of the same symbol in the object containing the reference must have a separate symbol table entry from the reference. Sections outside of the group can not reference symbols with STB_LOCAL binding for addresses that are contained in the group's sections, including symbols with type STT_SECTION.
- Non-symbol references to the sections comprising a group are not allowed from outside the group. For example, you cannot use a group member's section header index in an sh_link or sh_info member.
- A symbol table entry defined relative to one of the group's sections can be removed if the group members are discarded. This removal occurs if the symbol table entry is contained in a symbol table section that is not part of the group.

Capabilities Section

A SHT_SUNW_cap section identifies the capability requirements of an object. These capabilities are referred to as *object* capabilities. This section can also identify the capability requirements of functions within an object. These capabilities are referred to as *symbol* capabilities. This section contains an array of the following structures. See sys/elf.h.

```
typedef struct {
        Elf32_Word      c_tag;
        union {
                Elf32_Word      c_val;
                Elf32_Addr      c_ptr;
        } c_un;
} Elf32_Cap;

typedef struct {
        Elf64_Xword     c_tag;
        union {
                Elf64_Xword     c_val;
                Elf64_Addr      c_ptr;
        } c_un;
} Elf64_Cap;
```

For each object with this type, c_tag controls the interpretation of c_un.

c_val
: These objects represent integer values with various interpretations.

c_ptr
: These objects represent program virtual addresses.

The following capabilities tags exist.

TABLE 7–12 ELF Capability Array Tags

Name	Value	c_un
CA_SUNW_NULL	0	Ignored
CA_SUNW_HW_1	1	c_val
CA_SUNW_SF_1	2	c_val
CA_SUNW_HW_2	3	c_val
CA_SUNW_PLAT	4	c_ptr
CA_SUNW_MACH	5	c_ptr
CA_SUNW_ID	6	c_ptr

CA_SUNW_NULL
: Marks the end of a group of capabilities.

CA_SUNW_HW_1, CA_SUNW_HW_2
: Indicates hardware capability values. The c_val element contains a value that represents the associated hardware capabilities. On SPARC platforms, hardware capabilities are defined in sys/auxv_SPARC.h. On x86 platforms, hardware capabilities are defined in sys/auxv_386.h.

CA_SUNW_SF_1
: Indicates software capability values. The c_val element contains a value that represents the associated software capabilities that are defined in sys/elf.h.

CA_SUNW_PLAT
: Specifies a platform name. The c_ptr element contains the string table offset of a null-terminated string, that defines a platform name.

CA_SUNW_MACH
: Specifies a machine name. The c_ptr element contains the string table offset of a null-terminated string, that defines a machine hardware name.

CA_SUNW_ID
: Specifies a capability identifier name. The c_ptr element contains the string table offset of a null-terminated string, that defines an identifier name. This element does not define a capability, but allows a capability group to be identified. This identifier name is appended to

any function names that are transformed to local symbols as part of the link-editors `-z symbolcap` processing. See "Converting Object Capabilities to Symbol Capabilities" on page 77.

Relocatable objects can contain a capabilities section. The link-editor combines any capabilities sections from multiple input relocatable objects into a single capabilities section. The link-editor also allows capabilities to be defined at the time an object is built. See "Identifying Capability Requirements" on page 64.

Multiple `CA_SUNW_NULL` terminated groups of capabilities can exist within an object. The first group, starting at index `0`, identifies the object capabilities. A dynamic object that defines object capabilities, has a `PT_SUNWCAP` program header associated to the section. This program header allows the runtime linker to validate the object against the system capabilities that are available to the process. Dynamic objects that use different object capabilities can provide a flexible runtime environment using filters. See "Capability Specific Shared Objects" on page 379.

Additional groups of capabilities identify symbol capabilities. Symbol capabilities allow multiple instances of the same symbol to exist within an object. Each instance is associated to a set of capabilities that must be available for the instance to be used. When symbol capabilities are present, the `sh_link` element of the `SHT_SUNW_cap` section points to the associated `SHT_SUNW_capinfo` table. Dynamic objects that use symbol capabilities can provide a flexible means of enabling optimized functions for specific systems. See "Creating a Symbol Capabilities Family" on page 73.

The `SHT_SUNW_capinfo` table parallels the associated symbol table. The `sh_link` element of the `SHT_SUNW_capinfo` section points to the associated symbol table. Functions that are associated with capabilities, have indexes within the `SHT_SUNW_capinfo` table that identify the capabilities group within the `SHT_SUNW_cap` section.

Within a dynamic object, the `sh_info` element of the `SHT_SUNW_capinfo` section points to a capabilities chain table, `SHT_SUNW_capchain`. This table is used by the runtime linker to locate members of a capabilities family.

A `SHT_SUNW_capinfo` table entry has the following format. See `sys/elf.h`.

```
typedef Elf32_Word    Elf32_Capinfo;
typedef Elf64_Word    Elf64_Capinfo;
```

Elements within this table are interpreted using the following macros. See `sys/elf.h`.

```
#define ELF32_C_SYM(info)       ((info)>>8)
#define ELF32_C_GROUP(info)     ((unsigned char)(info))
#define ELF32_C_INFO(sym, grp)  (((sym)<<8)+(unsigned char)(grp))

#define ELF64_C_SYM(info)       ((info)>>32)
#define ELF64_C_GROUP(info)     ((Elf64_Word)(info))
#define ELF64_C_INFO(sym, grp)  (((Elf64_Xword)(sym)<<32)+(Elf64_Xword)(grp))
```

A SHT_SUNW_capinfo entry group element contains the index of the SHT_SUNW_cap table that this symbol is associated with. This element thus associates symbols to a capability group. A reserved group index, CAPINFO_SUNW_GLOB, identifies a lead function of a family of capabilities instances, that provides a default instance.

Name	Value	Meaning
CAPINFO_SUNW_GLOB	0xff	Identifies a default function, that is not associated with any symbol capabilities.

A SHT_SUNW_capinfo entry symbol element contains the index of the lead function associated with this symbol. The group and symbol information allow the link-editor to process families of capabilities symbols from relocatable objects, and construct the necessary capabilities information in any output object. Within a dynamic object, the symbol element of a lead symbol, one tagged with the group CAPINFO_SUNW_GLOB, is an index into the SHT_SUNW_capchain table. This index allows the runtime linker to traverse the capabilities chain table, starting at this index, and inspects each following entry until a 0 entry is found. The chain entries contain symbol indices for each capabilities family member.

A dynamic object that defines symbol capabilities, has a DT_SUNW_CAP dynamic entry, and a DT_SUNW_CAPINFO dynamic entry. These entries identify the SHT_SUNW_cap section, and SHT_SUNW_capinfo section respectively. The object also contains DT_SUNW_CAPCHAIN, DT_SUNW_CAPCHINENT and DT_SUNW_CAPCHAINSZ entries that identify the SHT_SUNW_capchain section, the sections entry size and total size. These entries allow the runtime linker to establish the best function to use, from a family of function capability instances.

An object can define only object capabilities, or can define only symbol capabilities, or can define both types of capabilities. An object capabilities group starts at index 0. Symbol capabilities groups start at any index other than 0. If an object defines symbol capabilities, but no object capabilities, then a single CA_SUNW_NULL entry must exist at index 0 to indicate the start of symbol capabilities.

Hash Table Section

A hash table consists of Elf32_Word or Elf64_Word objects that provide for symbol table access. The SHT_HASH section provides this hash table. The symbol table to which the hashing is associated is specified in the sh_link entry of the hash table's section header. Labels are used in the following figure to help explain the hash table organization, but these labels are not part of the specification.

TABLE 7–32 ELF Dynamic Array Tags *(Continued)*

Name	Value	d_un	Executable	Shared Object
DT_SUNW_SORTENT	0x60000013	d_val	Optional	Optional
DT_SUNW_SYMSORT	0x60000014	d_ptr	Optional	Optional
DT_SUNW_SYMSORTSZ	0x60000015	d_val	Optional	Optional
DT_SUNW_TLSSORT	0x60000016	d_ptr	Optional	Optional
DT_SUNW_TLSSORTSZ	0x60000017	d_val	Optional	Optional
DT_SUNW_CAPINFO	0x60000018	d_ptr	Optional	Optional
DT_SUNW_STRPAD	0x60000019	d_val	Optional	Optional
DT_SUNW_CAPCHAIN	0x6000001a	d_ptr	Optional	Optional
DT_SUNW_LDMACH	0x6000001b	d_val	Optional	Optional
DT_SUNW_CAPCHAINENT	0x6000001d	d_val	Optional	Optional
DT_SUNW_CAPCHAINSZ	0x6000001f	d_val	Optional	Optional
DT_HIOS	0x6ffff000	Unspecified	Unspecified	Unspecified
DT_VALRNGLO	0x6ffffd00	Unspecified	Unspecified	Unspecified
DT_CHECKSUM	0x6ffffdf8	d_val	Optional	Optional
DT_PLTPADSZ	0x6ffffdf9	d_val	Optional	Optional
DT_MOVEENT	0x6ffffdfa	d_val	Optional	Optional
DT_MOVESZ	0x6ffffdfb	d_val	Optional	Optional
DT_FEATURE_1	0x6ffffdfc	d_val	Optional	Optional
DT_POSFLAG_1	0x6ffffdfd	d_val	Optional	Optional
DT_SYMINSZ	0x6ffffdfe	d_val	Optional	Optional
DT_SYMINENT	0x6ffffdff	d_val	Optional	Optional
DT_VALRNGHI	0x6ffffdff	Unspecified	Unspecified	Unspecified
DT_ADDRRNGLO	0x6ffffe00	Unspecified	Unspecified	Unspecified
DT_CONFIG	0x6ffffefa	d_ptr	Optional	Optional
DT_DEPAUDIT	0x6ffffefb	d_ptr	Optional	Optional
DT_AUDIT	0x6ffffefc	d_ptr	Optional	Optional
DT_PLTPAD	0x6ffffefd	d_ptr	Optional	Optional
DT_MOVETAB	0x6ffffefe	d_ptr	Optional	Optional

DT_SUNW_RTLDINF
: Reserved for internal use by the runtime-linker.

DT_SUNW_FILTER
: The DT_STRTAB string table offset of a null-terminated string that names one or more per-symbol, standard filtees. See "Generating Standard Filters" on page 133.

DT_SUNW_CAP
: The address of the capabilities section. See "Capabilities Section" on page 243.

DT_SUNW_SYMTAB
: The address of the symbol table containing local function symbols that augment the symbols provided by DT_SYMTAB. These symbols are always adjacent to, and immediately precede the symbols provided by DT_SYMTAB. See "Symbol Table Section" on page 265.

DT_SUNW_SYMSZ
: The combined size of the symbol tables given by DT_SUNW_SYMTAB and DT_SYMTAB.

DT_SUNW_ENCODING
: Dynamic tag values that are greater than or equal to DT_SUNW_ENCODING, and less than or equal to DT_HIOS, follow the rules for the interpretation of the d_un union.

DT_SUNW_SORTENT
: The size, in bytes, of the DT_SUNW_SYMSORT and DT_SUNW_TLSSORT symbol sort entries.

DT_SUNW_SYMSORT
: The address of the array of symbol table indices that provide sorted access to function and variable symbols in the symbol table referenced by DT_SUNW_SYMTAB. See "Symbol Sort Sections" on page 273.

DT_SUNW_SYMSORTSZ
: The total size, in bytes, of the DT_SUNW_SYMSORT array.

DT_SUNW_TLSSORT
: The address of the array of symbol table indices that provide sorted access to thread local symbols in the symbol table referenced by DT_SUNW_SYMTAB. See "Symbol Sort Sections" on page 273.

DT_SUNW_TLSSORTSZ
: The total size, in bytes, of the DT_SUNW_TLSSORT array.

DT_SUNW_CAPINFO
: The address of the array of symbol table indices that provide the association of symbols to their capability requirements. See "Capabilities Section" on page 243.

DT_SUNW_STRPAD
: The total size, in bytes, of the unused reserved space at the end of the dynamic string table. If DT_SUNW_STRPAD is not present in an object, no reserved space is available.

DT_SUNW_CAPCHAIN
: The address of the array of capability family indices. Each family of indices is terminated with a 0 entry.

DT_SUNW_LDMACH
: The machine architecture of the link-editor that produced the object. DT_SUNW_LDMACH uses the same EM_ integer values used for the e_machine field of the ELF header. See "ELF Header" on page 212. DT_SUNW_LDMACH is used to identify the class, 32–bit or 64–bit, and the platform of the link-editor that built the object. This information is not used by the runtime linker, but exists purely for documentation.

DT_SUNW_CAPCHAINENT
: The size, in bytes, of the DT_SUNW_CAPCHAIN entries.

DT_SUNW_CAPCHAINSZ
: The total size, in bytes, or the DT_SUNW_CAPCHAIN chain.

DT_SYMINFO
: The address of the symbol information table. This element requires that the DT_SYMINENT and DT_SYMINSZ elements also be present. See "Syminfo Table Section" on page 276.

DT_SYMINENT
: The size, in bytes, of the DT_SYMINFO information entry.

DT_SYMINSZ
: The total size, in bytes, of the DT_SYMINFO table.

DT_VERDEF
: The address of the version definition table. Elements within this table contain indexes into the string table DT_STRTAB. This element requires that the DT_VERDEFNUM element also be present. See "Version Definition Section" on page 278.

DT_VERDEFNUM
: The number of entries in the DT_VERDEF table.

DT_VERNEED
: The address of the version dependency table. Elements within this table contain indexes into the string table DT_STRTAB. This element requires that the DT_VERNEEDNUM element also be present. See "Version Dependency Section" on page 280.

DT_VERNEEDNUM
: The number of entries in the DT_VERNEEDNUM table.

DT_RELACOUNT
: Indicates the RELATIVE relocation count, which is produced from the concatenation of all Elf32_Rela, or Elf64_Rela relocations. See "Combined Relocation Sections" on page 150.

DT_RELCOUNT
: Indicates the RELATIVE relocation count, which is produced from the concatenation of all Elf32_Rel relocations. See "Combined Relocation Sections" on page 150.

◆ ◆ ◆ APPENDIX C

Establishing Dependencies with Dynamic String Tokens

A dynamic object can establish dependencies explicitly or through filters. Each of these mechanisms can be augmented with a *runpath*, which directs the runtime linker to search for and load the required dependency. String names used to record filters, dependencies and runpath information can be augmented with the following reserved dynamic string tokens.

- `$CAPABILITY ($HWCAP)`
- `$ISALIST`
- `$OSNAME`, `$OSREL`, `$PLATFORM` and `$MACHINE`
- `$ORIGIN`

The following sections provide examples of how each of these tokens can be employed.

Capability Specific Shared Objects

The dynamic token `$CAPABILITY` can be used to specify a directory in which capability specific shared objects exist. This token is available for filters and dependencies. As this token can expand to multiple objects, its use with dependencies is controlled. Dependencies obtained with `dlopen(3C)`, can use this token with the mode `RTLD_FIRST`. Explicit dependencies that use this token will load the first appropriate dependency found.

Note – The original capabilities implementation was based solely on hardware capabilities. The token `$HWCAP` was used to select this capability processing. Capabilities have since been extended beyond hardware capabilities, and the `$HWCAP` token has been replaced by the `$CAPABILITY` token. For compatibility, the `$HWCAP` token is interpreted as an alias for the `$CAPABILITY` token.

The path name specification must consist of a full path name terminated with the `$CAPABILITY` token. Shared objects that exist in the directory that is specified with the `$CAPABILITY` token are inspected at runtime. These objects should indicate their capability requirements. See

"Identifying Capability Requirements" on page 64. Each object is validated against the capabilities that are available to the process. Those objects that are applicable for use with the process, are sorted in descending order of their capability values. These sorted filtees are used to resolve symbols that are defined within the filter.

Filtees within the capabilities directory have no naming restrictions. The following example shows how the auxiliary filter `libfoo.so.1` can be designed to access hardware capability filtees.

```
$ LD_OPTIONS='-f /opt/ISV/lib/cap/$CAPABILITY' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
  [1]    SONAME    libfoo.so.1
  [2]    AUXILIARY /opt/ISV/lib/cap/$CAPABILITY
$ elfdump -H /opt/ISV/lib/cap/*

/opt/ISV/lib/cap/filtee.so.3:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/cap/filtee.so.1:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/cap/filtee.so.2:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

If the filter `libfoo.so.1` is processed on a system where the `MMX` and `SSE` hardware capabilities are available, the following filtee search order occurs.

```
$ cc -o prog prog.c -R. -lfoo
$ LD_DEBUG=symbols prog
....
01233: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01233: symbol=foo;  lookup in file=cap/filtee.so.2  [ ELF ]
```

```
01233: symbol=foo;  lookup in file=cap/filtee.so.1  [ ELF ]
....
```

Note that the capability value for `filtee.so.2` is greater than the capability value for `filtee.so.1`. `filtee.so.3` is not a candidate for inclusion in the symbol search, as the `SSE2` capability is not available.

Reducing Filtee Searches

The use of `$CAPABILITY` within a filter enables one or more filtees to provide implementations of interfaces that are defined within the filter.

All shared objects within the specified `$CAPABILITY` directory are inspected to validate their availability, and to sort those found appropriate for the process. Once sorted, all objects are loaded in preparation for use.

A filtee can be built with the link-editor's `-z endfiltee` option to indicate that it is the last of the available filtees. A filtee identified with this option, terminates the sorted list of filtees for that filter. No objects sorted after this filtee are loaded for the filter. From the previous example, if the `filter.so.2` filtee was tagged with `-z endfiltee`, the filtee search would be as follows.

```
$ LD_DEBUG=symbols prog
....
01424: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01424: symbol=foo;  lookup in file=cap/filtee.so.2  [ ELF ]
....
```

Instruction Set Specific Shared Objects

The dynamic token `$ISALIST` is expanded at runtime to reflect the native instruction sets executable on this platform, as displayed by the utility `isalist(1)`. This token is available for filters, runpath definitions, and dependencies. As this token can expand to multiple objects, its use with dependencies is controlled. Dependencies obtained with `dlopen(3C)`, can use this token with the mode `RTLD_FIRST`. Explicit dependencies that use this token will load the first appropriate dependency found.

Note – This token is obsolete, and might be removed in a future release of Solaris. See "Capability Specific Shared Objects" on page 379 for the recommended technique for handling instruction set extensions.

Any string name that incorporates the `$ISALIST` token is effectively duplicated into multiple strings. Each string is assigned one of the available instruction sets.