# RM0273
# Reference manual

## ST200 ELF specification

### Synopsis

This document describes the use of the ELF file format for the ST200.

# Contents

# 1 Introduction

This document describes the use of the ELF file format for the ST200 processor. It provides information needed to create and interpret ELF files and will be of interest to anyone who has to design a system that creates or interprets ST200 ELF files.

This document does not define operating system interfaces or any conventions specific to any single operating system: OS-specific documentation must be consulted for this information.

## 1.1 How to use the ST200 ELF specification

The ELF file format specification is contained in two chapters of the *Unix System V generic ABI (gABI)*; namely, *Chapter 4 Object Files* and *Chapter 5 Program Loading and Dynamic Linking*. The *Unix System V generic ABI* specification necessarily omits processor-specific characteristics.

The ST200 ELF Specification provides the processor-specific characteristics for the ST200 processor that are referenced by the generic specification. Therefore, the generic System V ABI is the prime reference document, and this document supplements that specification.

## 1.2 Additional documents

The following documents are referenced from this specification:

- *ST231 core and instruction set architecture manual* (7645929)
- *ST200 runtime architecture specification* (7521848)

The *Unix System V generic ABI* specification may be found at www.sco.com/developers/devspecs.

# 2 Object files

This chapter describes the ELF file format for object files.

## 2.1 ELF header

This section describes the ELF file format requirements for machine information.

### 2.1.1 File class

The file class must be `ELFCLASS32`.

### 2.1.2 Data encoding

ST200 objects may use either the big endian (`ELFDATA2MSB`) or little endian (`ELFDATA2LSB`) data encoding. Files using the `ELFDATA2MSB` encoding may not be combined with objects using the `ELFDATA2LSB` encoding.

### 2.1.3 Operating system identification

The `e_ident[EI_OSABI]` value identifies the operating system and ABI to which the object is targeted, see *Table 1*.

**Table 1. Operating system identification, e_ident[EI_OSABI]**

| Name | Value | Meaning |
|---|---|---|
| `ELFOSABI_NONE` | `0` | OS ABI is unspecified |
| `ELFOSABI_LINUX` | `3` | GNU/Linux |
| `ELFOSABI_OS21` | `64` | STMicroelectronics OS21 |

### 2.1.4 Processor identification

Processor identification resides in the ELF header's e_machine member and may take the value listed in *Table 2*.

*Note: This value is defined in the generic ELF specification, it is repeated here for information.*

**Table 2. Processor identification, e_machine**

| Name | Value | Meaning |
|---|---|---|
| `EM_ST200` | `100` | STMicroelectronics ST200 |

### 2.1.5 Processor-specific flags

The ELF header `e_flags` member holds flags associated with the file, as listed in *Table 3*.The values of the fields are listed in *Table 4* through *Table 6*.

**Table 3. Field layout within e_flags**

| Name | Bit(s) | Meaning |
|---|---|---|
| ELF_ST200_ABI | [0:6] | Target ABI |
| ELF_ST200_CORE | [8:15] | ST200 core identification |
| ELF_ST200_CUT | [16:19] | ST200 silicon implementation identification |

**Table 4. Values for ELF_ST200_ABI**

| Name | Value | Meaning |
|---|---|---|
| ELF_ST200_ABI_NO | 0 | Target ABI is unspecified |
| ELF_ST200_ABI_EMBED | 2 | ST200 embedded ABI |
| ELF_ST200_ABI_PIC | 3 | ST200 PIC ABI |
| ELF_ST200_ABI_GCC | 4 | ST200 gcc ABI |
| ELF_ST200_ABI_UNDEF | 5 | Undefined |
| ELF_ST200_ABI_RELOC_EMBED | 6 | ST200 relocatable-embedded ABI |
| ELF_ST200_ABI_MASK | 7f | All bits in this mask are reserved for ABI specific values |

**Table 5. Values for ELF_ST200_CORE**

| Name | Value | Meaning |
|---|---|---|
| ELF_ST200_CORE_ST210 (1) | 0 | Code targets ST210 core |
| ELF_ST200_CORE_ST220 (1) | 1 | Code targets ST220 core |
| ELF_ST200_CORE_ST230 (1) | 2 | Code targets ST230 core |
| ELF_ST200_CORE_ST231 | 3 | Code targets ST231 core |
| ELF_ST200_CORE_ST240 (1) | 6 | Code targets ST240 core |
| ELF_ST200_CORE_MASK | 0xff00 | All bits in this mask are reserved for core specific values |

1. This core is no longer supported by the current ST200 Micro Toolset.

**Table 6. Values for ELF_ST200_CUT**

| Name | Value | Meaning |
|---|---|---|
| ELF_ST200_CUT_0 | 0 | Cut 0 |
| ELF_ST200_CUT_1 | 1 | Cut 1 |
| ELF_ST200_CUT_2 | 2 | Cut 2 |
| ELF_ST200_CUT_3 | 3 | Cut 3 |

Table 6. Values for ELF_ST200_CUT (continued)

| Name | Value | Meaning |
|---|---|---|
| ELF_ST200_CUT_4 | 4 | Cut 4 |
| ELF_ST200_CUT_5 | 5 | Cut 5 |
| ELF_ST200_CUT_MASK | 0xf0000 | All bits in this mask are reserved for cut-specific values |

## 2.2 Sections

This section describes the ELF file format requirements for sections.

### 2.2.1 Section types

No processor-specific section types are defined.

### 2.2.2 Section attribute flags

No processor-specific section attribute flags are defined.

### 2.2.3 Special sections

No processor-specific sections are defined.

## 2.3 Symbol table

Two bits of the st_other field are used to indicate additional properties of a symbol. They are intended for use by link-time optimizations:

```
#define STO_MOVEABLE(o) ((o) & 0x10)
#define STO_USED(o) ((o) & 0x20)
```

STO_MOVEABLE — When set, indicates that the symbol can be deleted or reordered relative to other symbols at link-time. Relocations against an STO_MOVEABLE symbol must not be resolved before the final link. Furthermore, relocations of addresses in the range [*sym, sym + size - 1*], (where *sym* is a symbol with the STO_MOVEABLE bit set) must not be resolved before the final link.

STO_USED — When set, indicates that the symbol must not be deleted at link-time.

## 2.4 Relocations

This section describes the ELF file format requirements for relocations.

### 2.4.1 Relocation types

*Figure 1* illustrates the operation and data fields that can be relocated. *Table 7* describes the data fields.

The byte ordering for these fields is specified by the `e_ident[EI_DATA]` value in the ELF header: `ELFDATA2LSB` means little-endian, `ELFDATA2MSB` means big-endian.

**Figure 1. Relocatable fields**

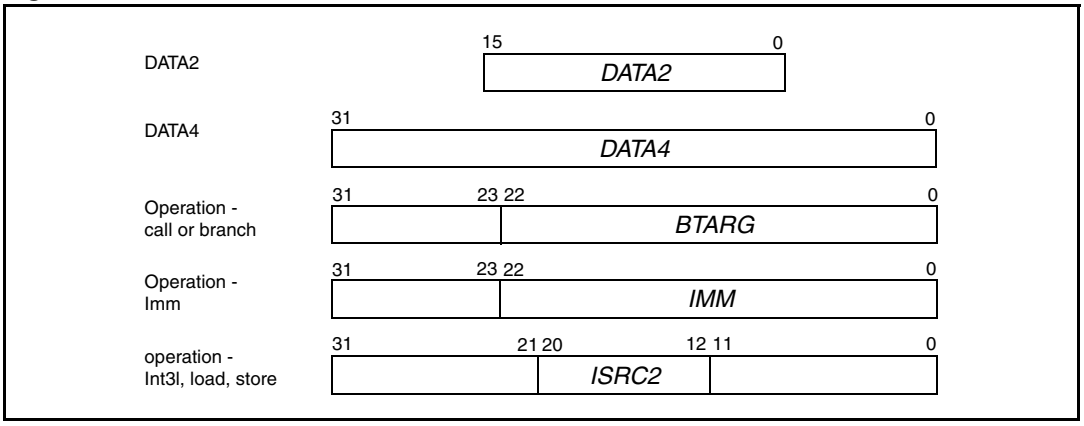



**Table 7. Data fields in *Figure 1***

| Data field | Description |
|---|---|
| *DATA2* | A 16-bit field occupying two bytes with arbitrary alignment. |
| *DATA4* | A 32-bit field occupying four bytes with arbitrary alignment. |
| *BTARG* | The *BTARG* field of an operation with call or branch format (see instruction encoding in the *ST2xx Core and Instruction Set Architecture Manual)*. |
| *IMM* | The *IMM* field of an operation (see instruction encoding in the *ST2xx Core and Instruction Set Architecture Manual*). |
| *ISRC2* | The *ISRC2* field of an operation (see instruction encoding in *ST2xx Core and Instruction Set Architecture Manual*). |

*Table 8* lists the relocation types. A description of the notation used in *Table 8* is given in *Table 9 on page 9*.

**Table 8. ST200 relocation types**

| Name | Value | Field | Calculation |
|---|---|---|---|
| `R_ST200_NONE` | 0 | None | None |
| `R_ST200_16` | 1 | *DATA2* | *S* + *A* |
| `R_ST200_32` | 2 | *DATA4* | *S* + *A* |
| `R_ST200_32_PCREL` | 3 | *DATA4* | *S* + *A* – *P* |

**Table 8. ST200 relocation types (continued)**

| Name | Value | Field | Calculation |
|---|---|---|---|
| R_ST200_23_PCREL | 4 | *BTARG* | (*S* + *A* – *P*) >> 2 |
| R_ST200_HI23 | 5 | *IMM* | (*S* + *A*) >> 9 |
| R_ST200_LO9 | 6 | *ISRC2* | (*S* + *A*) & 0x1ff |
| R_ST200_GPREL_HI23 | 7 | *IMM* | (@gprel(*S* + *A*)) >> 9 |
| R_ST200_GPREL_LO9 | 8 | *ISRC2* | (@gprel(*S* +*A*)) & 0x1ff |
| R_ST200_REL32 | 9 | *DATA4* | *BD* + *A* |
| R_ST200_GOTOFF_HI23 | 10 | *IMM* | (@gotoff(*S* + *A*)) >> 9 |
| R_ST200_GOTOFF_LO9 | 11 | *ISRC2* | (@gotoff(*S* + *A*)) & 0x1ff |
| R_ST200_LTV32 | 14 | *DATA4* | @ltv(*S* + *A*), see *Table 9*. |
| R_ST200_SEGREL32 | 15 | *DATA4* | @segrel(*S* + *A*) |
| R_ST200_NEGGPREL_HI23 | 22 | *IMM* | (@neggprel(*S* + *A*)) >> 9 |
| R_ST200_NEGGPREL_LO9 | 23 | *ISRC2* | (@neggprel(*S* + *A*)) & 0x1ff |
| R_ST200_COPY | 24 | None | None, see *Table 9*. |
| R_ST200_JMP_SLOT | 25 | *DATA4* | *S*, see *Table 9*. |
| R_ST200_TPREL_HI23 | 26 | *IMM* | (@tprel(*S* + *A*)) >> 9 |
| R_ST200_TPREL_LO9 | 27 | *ISRC2* | (@tprel(*S* + *A*)) & 0x1ff |
| R_ST200_TPREL32 | 28 | *DATA4* | @tprel(*S* + *A*) |
| R_ST200_GOTOFF_TPREL_HI23 | 29 | *IMM* | (@gotoff(@tprel(*S* + *A*)))>>9 |
| R_ST200_GOTOFF_TPREL_LO9 | 30 | *ISRC2* | (@gotoff(@tprel(*S* + *A*))) & 0x1ff |
| R_ST200_GOTOFF_DTPLDM_HI23 | 31 | *IMM* | (@gotoff(@dtpldm(*S* + *A*)))>>9 |
| R_ST200_GOTOFF_DTPLDM_LO9 | 32 | *ISRC2* | (@gotoff(@dtpldm(*S* + *A*)))& 0x1ff |
| R_ST200_DTPREL_HI23 | 33 | *IMM* | (@dtprel(*S* + *A*))>>9 |
| R_ST200_DTPREL_LO9 | 34 | *ISRC2* | (@dtprel(*S* + *A*))& 0x1ff |
| R_ST200_DTPMOD32 | 35 | *DATA4* | @dtpmod(*S* + *A*) |
| R_ST200_DTPREL32 | 36 | *DATA4* | @dtprel(*S* + *A*) |
| R_ST200_GOTOFF_DTPNDX_HI23 | 37 | *IMM* | (@gotoff(@dtpndx(*S* + *A*)))>>9 |
| R_ST200_GOTOFF_DTPNDX_LO9 | 38 | *ISRC2* | (@gotoff(@dtpndx(*S* + *A*)))& 0x1ff |

**Table 9. Notation used in *Table 8***

| Notation | Meaning |
|---|---|
| *A* | The *addend* used to compute the value of the relocatable field. Present in the relocation for `.rela` relocations, or in the storage unit being relocated for `.rel` relocations. |
| *BD* | The *base* address *difference*, a constant that must be applied to a virtual address. This constant represents the difference between the run-time address and the link-time address of a segment. |

**Table 9. Notation used in *Table 8* (continued)**

| Notation | Meaning |
|---|---|
| *GP* | The global pointer value. This value is selected by the link editor, it is an address at a constant offset from the base of the global offset table (GOT). (Normally, the constant offset is 0, but this is not a requirement of this specification.) |
| *P* | The *place* (section offset or address) of the storage unit being relocated. |
| *S* | The value of the *symbol* whose index resides in the relocation entry. |
| `@dtpldm`(*expr*) | Computes a `ti_index` structure for *expr*. The `ti_index` structure contains two word-sized fields: the first contains the dynamic module identity (id) for the module containing *expr*, the second contains 0. If *expr* is an absolute value rather than a symbol relative value, then the dynamic module id is set to the current module. |
| `@dtpmod`(*expr*) | Computes the dynamic module id for the module containing *expr*. |
| `@dtpndx`(*expr*) | Computes a `ti_index` structure for *expr*. The `ti_index` structure contains two word-sized fields: the first contains the dynamic module id for the module containing *expr*, the second contains the offset of *expr* from the start of the thread-local storage (TLS) block that contains it. If *expr* is an absolute value rather than a symbol relative value, then the dynamic module id is set to the current module. |
| `@dtprel`(*expr*) | Computes the offset of *expr* from the start of the TLS block that contains *expr*. |
| `@gotoff`(*expr*) | Requests the creation of a global offset table entry that will hold the full value of *expr*, and computes the *gp*-relative displacement to that global offset table entry. |
| `@gprel`(*expr*) | Computes a *gp*-relative displacement: the difference between *expr* and the global pointer value for the current module. |
| `@ltv`(*expr*) | Represents the "*link-time value*" of the expression. This is the value of the expression at link-time. At run-time, the value of *expr* may differ, because the dynamic object will not necessarily be loaded at the address the link editor assumed at link-time, and so symbols in the expression may have different values. However, there is a constant offset between the link-time value and the run-time value, this constant is the base address difference (*BD*), so the run-time value of a symbol can be represented as `@ltv(symbol)` + *BD*. |
| `@neggprel`(*expr*) | Computes the negative of `@gprel(`*expr*`)`, that is, the difference between the global pointer value for the current module and *expr*. |
| `@segrel`(*expr*) | Computes a segment-relative displacement: the difference between *expr* and the address of the beginning of the segment containing the object to be relocated. This relocation type is designed for debug tables or unwind tables that need to contain pointers but remain shareable. |
| `@tprel`(*expr*) | Computes a *tp*-relative displacement: the difference between *expr* and the thread pointer value for the current module. Note that this displacement is always positive, because thread data areas are laid out at a more positive address than the *TP* value. |

### R_ST200_GPREL_HI23
### R_ST200_GPREL_LO9

These relocation types should only be used on operations where the *SRC1* operand is a register that contains the *GP* value. This allows the link editor to rewrite the operation to be an absolute reference, see *Transformation of GP-relative code to absolute code on page 13*.

### R_ST200_GOTOFF_HI23
### R_ST200_GOTOFF_LO9

These relocation types compute the distance from the *GP* value to the symbol's global offset table (GOT) entry. They instruct the link editor to create a global offset table, if one has not already been created. They also instruct the link editor to add an entry for the symbol (plus its addend) to the GOT, if one does not already exist.

These relocation types should only be used on operations where the *SRC1* operand is a register that contains the *GP* value. This allows the link editor to rewrite the operation to be an absolute reference, see *Transformation of GP-relative code to absolute code on page 13*.

### R_ST200_LTV32

This relocation type appears only in relocatable objects. It behaves identically to `R_ST200_32` (see *Table 8 on page 8*), with one exception: while it is expected that the addresses created will need a further relocation at run-time, the link editor should not create a corresponding dynamic relocation in the output executable or shared object file. The run-time consumer of the information provided is expected to relocate these values.

### R_ST200_COPY

The link editor creates this relocation type for dynamic linking. It's offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies the initialized value of the symbol from the shared object to the location specified by the offset.

### R_ST200_JMP_SLOT

The link editor creates this relocation type for dynamic linking. It's offset member gives the location of a global offset table entry. The dynamic linker converts the global offset table entry to contain the designated symbol's address. This relocation type differs from `R_ST200_32` in that the dynamic linker is allowed to use lazy binding on an `R_ST200_JMP_SLOT` relocation.

### R_ST200_TPREL_HI23
### R_ST200_TPREL_LO9

These relocation types should only be used on operations where the *SRC1* operand is a register that contains the *TP* value.

*S* must be a thread variable.

### R_ST200_GOTOFF_TPREL_HI23
### R_ST200_GOTOFF_TPREL_LO9

These relocations instruct the link editor to create a global offset table (GOT), if one has not already been created. They also instruct the link editor to add an entry containing the value

`@tprel(S+A)` to the GOT, if one does not already exist. The link editor creates a dynamic relocation of type `R_ST200_TPREL32` to initialize the global offset table entry.

*S* must be a thread variable.

These relocation types should only be used on operations where the *SRC1* operand is a register that contains the *GP* value. This allows the link editor to rewrite the operation to be an absolute reference, see *Transformation of GP-relative code to absolute code on page 13*.

### R_ST200_GOTOFF_DTPLDM_HI23
### R_ST200_GOTOFF_DTPLDM_LO9

These relocations instruct the link editor to create a global offset table, if one has not already been created. They also instruct the link editor to add a `ti_index` structure for the expression *S+A* (where *S* is a thread variable) to the GOT, if one does not already exist. The link editor creates a dynamic relocation of type `R_ST200_DTPMOD32` to initialize the `ti_module` field of the `ti_index` structure, and a dynamic relocation of type `R_ST200_DTPREL32` to initialize the `ti_offset` field of the `ti_index` structure.

These relocation types should only be used on operations where the *SRC1* operand is a register that contains the `GP` value. This allows the link editor to rewrite the operation to be an absolute reference, see *Transformation of GP-relative code to absolute code*.

### R_ST200_GOTOFF_DTPNDX_HI23
### R_ST200_GOTOFF_DTPNDX_LO9

These relocations instruct the link editor to create a GOT, if one has not already been created. They also instruct the link editor to add a `ti_index` structure for the base of the module containing the symbol *S* to the global offset table, if there is not already such an entry in the global offset table. The link editor creates a dynamic relocation of type `R_ST200_DTPMOD32` to initialize the `ti_module` field of the `ti_index` structure. The `ti_offset` field of the `ti_index` structure is set to 0.

These relocation types should only be used on operations where the *SRC1* operand is a register that contains the *GP* value. This allows the link editor to rewrite the operation to be an absolute reference, see *Transformation of GP-relative code to absolute code*.

## 2.4.2 Relocation transformations

This section lists some transformations that the link editor is permitted to perform on relocations. A link editor for ST200 is not required to perform these transformations: they provide optimization or additional functionality over a basic implementation.

### Optimization of GOT load

The link editor may transform a `load` from the global offset table into a *GP*-relative `add` provided that **all** the following conditions apply.

1. The original relocations are `R_ST200_GOTOFF_LO9` and `R_ST200_GOTOFF_HI23`.
2. The original relocations, relocate an `ldw` or `ldw.d` operation and its adjacent immediate extension[(a)].
3. Both relocations refer to the same symbol and have the same addend.
4. The symbol referred to is “own”, (it is defined in the current link and is not weak or preemptible).

The transformation is as follows.

1. Rewrite the `R_ST200_GOTOFF_LO9` relocation as `R_ST200_GPREL_LO9`, and the `R_ST200_GOTOFF_HI23` relocation as `R_ST200_GPREL_HI23`.
2. Rewrite the load operation as `add` immediate.

If the transformation is performed, then the transformed relocations do not require that a GOT entry is created for the symbol. After all relocations have been transformed, if there are no relocations that require a GOT entry for the symbol, then the GOT entry need not be created.

### Transformation of GP-relative code to absolute code

The link editor may transform a *GP*-relative symbol reference to an absolute reference, provided that **all** the following conditions apply.

1. Either the symbol referred to by the relocation is an absolute symbol, or the link editor is producing an absolute object.
2. The original relocations are one of the pairs:
   - `R_ST200_GPREL_LO9` and `R_ST200_GPREL_HI23`
   - `R_ST200_GOTOFF_LO9` and `R_ST200_GOTOFF_HI23`
   - `R_ST200_GOTOFF_TPREL_LO9` and `R_ST200_GOTOFF_TPREL_HI23`
   - `R_ST200_GOTOFF_DTPLDM_LO9` and `R_ST200_GOTOFF_DTPLDM_HI23`
   - `R_ST200_GOTOFF_DTPNDX_LO9` and `R_ST200_GOTOFF_DTPNDX_HI23`.
3. The original relocations relocate an operation and its immediate extension.
4. Both relocations refer to the same symbol and have the same addend.

The transformation is as follows:

1. Rewrite the *ISRC1* field of the operation relocated by the `R_ST200_xxx_LO9` relocation to be R0.
2. Modify the relocation calculation as described in *Table 10*.

a. The load operation must be unconditional, that is, the relocations must not relocate a conditional load operation.

**Table 10. Modified relocation calculation when transformed to absolute**

| Relocation | Modified calculation when transformed to absolute |
|---|---|
| R_ST200_GPREL_HI23 | (*S* + *A*) >> 9 |
| R_ST200_GPREL_LO9 | (*S* + *A*) & 0x1ff |
| R_ST200_GOTOFF_HI23 | (*GP* + @gotoff(*S* + *A*)) >> 9 |
| R_ST200_GOTOFF_LO9 | (*GP* + @gotoff(*S* + *A*)) & 0x1ff |
| R_ST200_GOTOFF_TPREL_HI23 | (*GP* + @gotoff(@tprel(*S*+*A*)))>>9 |
| R_ST200_GOTOFF_TPREL_LO9 | (*GP* + @gotoff(@tprel(*S*+*A*))) & 0x1ff |
| R_ST200_GOTOFF_DTPLDM_HI23 | (*GP* + @gotoff(@dtpldm(*S*+*A*)))>>9 |
| R_ST200_GOTOFF_DTPLDM_LO9 | (*GP* + @gotoff(@dtpldm(*S*+*A*)))& 0x1ff |
| R_ST200_GOTOFF_DTPNDX_HI23 | (*GP* + @gotoff(@dtpndx(*S*+*A*)))>>9 |
| R_ST200))_GOTOFF_DTPNDX_LO9 | (*GP* + @gotoff(@dtpndx(*S*+*A*)))&0x1ff |

*Note: When the R_ST200_GOTOFF_xxxx relocations are transformed to absolute, they still require a GOT entry containing the value of the symbol + addend. After the transformation, this GOT entry is accessed using absolute addressing instead of GP-relative addressing.*

The GOT load optimization (see *Optimization of GOT load*) and the absolute addressing transformation (this section) may both be applied to the same pair of relocations as illustrated in the following example.

### Example

The compiler is generating PIC code and needs the value or address of var, a symbol defined in another compilation unit. As the compiler does not know if the symbol is “own”, it must load the address of the symbol from the GOT, so it generates the following operations:

| Operation | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_GOTOFF_HI23 | var | 0 |
| ldw t1=0[$r14] | R_ST200_GOTOFF_LO9 | var | 0 |

At link time, the link editor has visibility of the entire load module, and determines that var is an “own” symbol. It therefore performs the GOT load transformation (see *Optimization of GOT load on page 13*) on the operations and relocations, producing:

| Operation | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_GPREL_HI23 | var | 0 |
| add t1=$r14,0 | R_ST200_GPREL_LO9 | var | 0 |

This avoids the need for the load operation. (If all GOTOFF relocations that reference var are transformed in this way it also avoids the need for a global offset table entry for var.)

If the link editor is producing an absolute object, it may also apply the absolute addressing transformation to this code, so that the final code emitted is:

| Operation |
|---|
| imml (var >> 9) |
| add t1=$r0,(var & 0x1ff) |

### Transforming from general dynamic to initial exec

A code sequence that uses the general dynamic thread-local storage (TLS) model may be transformed to the initial exec TLS model, provided that **all** the following conditions apply.

1. There is an adjacent pair of R_ST200_GOTOFF_DTPNDX_LO9 and R_ST200_GOTOFF_DTPNDX_HI23 relocations.
2. These relocations relocate an **add** operation and its immediate extension.
3. These relocations refer to the same symbol and have the same addend.
4. That symbol is defined in the current module or one of the shared libraries that are inputs to the link.
5. The static access TLS model is allowed (by a command-line option).
6. In the same bundle as the load operation, there is a R_ST200_PCREL23 relocation relocating a **call** operation, with a symbol named __tls_get_addr and an offset of zero.
7. The add operation sets R16.
8. The target processor has pipeline interlocks.

The transformation is as follows.

1. Rewrite the R_ST200_GOTOFF_DTPNDX_LO9/R_ST200_GOTOFF_DTPNDX_HI23 relocations to be R_ST200_GOTOFF_TPREL_LO9/R_ST200_GOTOFF_TPREL_HI23.
2. Rewrite the **add** instruction to be **ldw**.
3. Rewrite the **call** instruction to be **add $r16=$r13,$r16**, and rewrite the R_ST200_PCREL23 relocation to be R_ST200_NONE.

#### Example

Before:

**Table 11. General dynamic to initial exec, before**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_GOTOFF_DTPNDX_HI23 | *x* | 0 |
| add $r16=$r14,@gotoff(@dtpndx(x));; | R_ST200_GOTOFF_DTPNDX_LO9 | *x* | 0 |
| call $r63=__tls_get_addr ;; | R_ST200_23_PCREL | __tls_get_addr | 0 |

After:

**Table 12. ST200 General dynamic to initial exec, after**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_GOTOFF_TPREL_HI23` | *x* | 0 |
| `ldw $r16=$r14,@gotoff(@tprel(x));;` | `R_ST200_GOTOFF_TPREL_LO9` | *x* | 0 |
| `add $r16=$r13,$r16 ;;` | `R_ST200_NONE` | | |

### Transforming from general dynamic to local exec

A code sequence that uses the general dynamic TLS model may be transformed to the local exec TLS model, provided that **all** the following conditions apply.

1. There is an adjacent pair of `R_ST200_GOTOFF_DTPNDX_LO9` and `R_ST200_GOTOFF_DTPNDX_HI23` relocations.
2. These relocations relocate an **add** operation and its immediate extension.
3. These relocations refer to the same symbol and have the same addend.
4. That symbol is defined and not weak or preemptible.
5. In the same bundle as the add operation, there is a `R_ST200_PCREL23` relocation relocating a **call** operation, with a symbol named `__tls_get_addr` and an offset of zero.
6. The add operation sets R16.
7. The linker is producing a main program.

The transformation is to:

1. Rewrite the `R_ST200_GOTOFF_DTPNDX_LO9` and `R_ST200_GOTOFF_DTPNDX_HI23` relocations to be `R_ST200_TPREL_LO9` and `R_ST200_TPREL_HI23`.
2. Rewrite the *SRC1* field of the **add** instruction to be **R13**.
3. Rewrite the **call** instruction to be **nop**, and rewrite the `R_ST200_PCREL23` relocation to be `R_ST200_NONE`.

#### Example

Before:

**Table 13. General dynamic to local exec, before**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_GOTOFF_DTPNDX_HI23` | *x* | 0 |
| `add $r16=$r14,@gotoff(@dtpndx(x));;` | `R_ST200_GOTOFF_DTPNDX_LO9` | *x* | 0 |
| `call $r63=__tls_get_addr ;;` | `R_ST200_23_PCREL` | `__tls_get_addr` | 0 |

After:

**Table 14. ST200 General dynamic to local exec, after**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_TPREL_HI23` | *x* | 0 |
| `add $r16=$r13,@tprel(x);;` | `R_ST200_TPREL_LO9` | *x* | 0 |
| `nop ;;` | `R_ST200_NONE` | | |

### Transforming from local dynamic to local exec

A code sequence that uses the local dynamic TLS model may be transformed to the local exec TLS model, provided that **all** the following conditions apply.

1. There is an adjacent pair of `R_ST200_GOTOFF_DTPLDM_LO9` and `R_ST200_GOTOFF_DTPLDM_HI23` relocations.
2. These relocations relocate an **add** operation and its immediate extension.
3. These relocations refer to the same symbol and have the same addend.
4. That symbol is defined and not weak or preemptible.
5. In the same bundle as the add operation, there is a `R_ST200_PCREL23` relocation relocating a **call** operation, with a symbol named `__tls_get_addr` and an offset of zero.
6. The add operation sets R16.
7. The linker is producing a main program.

The transformation is as follows.

1. Rewrite the `R_ST200_GOTOFF_DTPLDM_LO9`/`R_ST200_GOTOFF_DTPLDM_HI23` relocations to be `R_ST200_TPREL_LO9`/`R_ST200_TPREL_HI23`.
2. Rewrite the symbol field of these two relocations to be a symbol representing the base of the TLS block containing the original symbol.
3. Rewrite the *SRC1* field of the **add** instruction to be **R13**.
4. Rewrite the **call** instruction to be **nop**, and rewrite the `R_ST200_PCREL23` relocation to be `R_ST200_NONE`.

#### Example

Before:

**Table 15. Local dynamic to local exec, before**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_GOTOFF_DTPLDM_HI23` | *x* | 0 |
| `add $r16=$r14,@gotoff(@dtpldm(x));;` | `R_ST200_GOTOFF_DTPLDM_LO9` | *x* | 0 |
| `call $r63=__tls_get_addr ;;` | `R_ST200_23_PCREL` | `__tls_get_addr` | 0 |
| `...` | | | |

After:

**Table 16. Local dynamic to local exec, after**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_TPREL_HI23` | Base of TLS block containing *x* | 0 |
| `add $r16=$r13,@tprel(x);;` | `R_ST200_TPREL_LO9` | Base of TLS block containing *x* | 0 |
| `nop ;;` | `R_ST200_NONE` | | |
| `...` | | | |

### Transforming from initial exec to local exec

A code sequence that uses the local dynamic TLS model may be transformed to the local exec TLS model, provided that **all** the following conditions apply.

1. There is an adjacent pair of `R_ST200_GOTOFF_TPREL_LO9` and `R_ST200_GOTOFF_TPREL_HI23` relocations.
2. These relocations relocate a **ldw** or **ldw.d** operation and its immediate extension.
3. These relocations refer to the same symbol and have the same addend.
4. That symbol is defined and not weak or preemptible.
5. The linker is producing a main program.

The transformation is as follows.

1. Rewrite the `R_ST200_GOTOFF_TPREL_LO9`/`R_ST200_GOTOFF_TPREL_HI23` relocations to be `R_ST200_TPREL_LO9`/`R_ST200_TPREL_HI23`.
2. Rewrite the **ldw** and **ldw.d** operations to be **mov** (**add** immediate with **R0** in the *SRC1* field).

#### Example

Before:

**Table 17. Initial exec to local exec, before**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_GOTOFF_TPREL_HI23` | *x* | 0 |
| `ldw $r23=$r14,@gotoff(@tprel(x));;` | `R_ST200_GOTOFF_TPREL_LO9` | *x* | 0 |
| `...` | | | |
| `add $r24=$r13, $r23;;` | | | |

After:

**Table 18. Initial exec to local exec, after**

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_TPREL_HI23` | *x* | 0 |
| `mov $r23=@tprel(x);;` | `R_ST200_TPREL_LO9` | *x* | 0 |
| `...` | | | |
| `add $r24=$r13, $r23;;` | | | |

# 3 Program loading and dynamic linking

This chapter describes the implementation of program loading and dynamic linking.

## 3.1 Program header

No processor-specific segment types are defined for the `p_type` field of the program header.

No processor-specific segment flags are defined for the `p_flags` field of the program header.

## 3.2 Program loading

Absolute executables contain absolute code, and must reside at the addresses (virtual or physical) specified when the executable was linked.

Shared objects typically contain position-independent code. This allows the object to be loaded at an arbitrary address, and allows the address to change from one process to another, without invalidating execution behaviour. For these objects, the virtual address (`p_vaddr`) field in a segment header might not represent the actual address of the object's memory image.

The ST200 runtime architecture permits position-independent code to use relative addressing between segments. This means that any difference between segment addresses in memory must match the difference between segment addresses calculated at link time, that is, the individual segments for a given executable or shared object are fixed relatively in relation to one another. The difference between the address of any segment in memory and the corresponding address in the file is a fixed constant value for every segment in a load module. This difference is the base address difference *BD*.

An ST200 program or shared object built to run in a virtual memory environment must satisfy the following rules:

1. The preferred page size for virtual memory management purposes for an ST200 segment is contained in the `p_align` field of the program header entry describing that segment.
2. Virtual addresses and file offsets for ST200 segments are congruent modulo the value contained in the `p_align` field.

## 3.3 Dynamic linking

When building an executable file that uses dynamic linking, the link editor adds a program header element of type `PT_INTERP` to the executable file, which provides the system with the path to the dynamic linker. The location of the dynamic linker is defined by the operating system.

### 3.3.1 Dynamic section

No processor-specific dynamic section tags are defined.

All dynamic section entries containing addresses (entries that use the `d_ptr` member) contain link-time addresses. The dynamic loader must relocate these addresses using the base address displacement.

### 3.3.2 Global offset table

In general, position-independent code cannot contain absolute addresses. A *global offset table* (GOT) is used to hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of the program text. Position-independent code references its global offset table using the *GP* value, and extracts the absolute addresses. The *GP* value is calculated using position-independent addressing. More details may be found in the *ST200 Run-time Architecture Manual*.

Initially the global offset table holds information as required by its relocation entries (see *Section 2.4.1: Relocation types on page 8*). After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which specify the values to place in the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory locations to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can therefore calculate the absolute addresses of all symbols in all segments.

If position-independent code requires direct access to the absolute address of a symbol, that symbol is given a global offset table entry. The executable file and each shared object have separate global offset tables, so a symbol's address may appear in several tables. The dynamic linker processes the global offset table relocations before giving control to the code, ensuring that absolute addresses are available during execution.

The system may choose different load addresses for the same shared object in different programs; it may even choose different load addresses for different executions of the same program.

Memory segments cannot be moved once the process image is established. As long as a process exists, its segments reside at fixed addresses.

### 3.3.3 Procedure linkage table

Operations which transfer control from one executable or shared object to another (such as function calls) are normally coded by the compiler as *PC*-relative calls and branches. The link editor cannot directly resolve these operations, because the relative offsets between load modules are not determined until load time. Furthermore, global symbols may be preempted, so it may not be possible even to determine the load module that defines a global symbol until load time.

The link editor creates an entry in the global offset table to hold the absolute address of the symbol, and a dynamic relocation to instruct the dynamic linker to initialize this global offset table entry.

The link editor also creates an import stub for the symbol and adds this to the *procedure linkage table* (PLT). This is a small piece of code that loads the absolute address of the symbol from the global offset table, and then performs an indirect jump to that address.

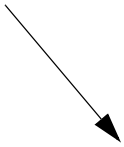
The PLT contains read-only code which is added to the object's shared text. The original call or branch operation is resolved to transfer control to the import stub, which then indirectly transfers control to the function.

The compiler may choose to compile the import stub in line. In this case, the link editor does not need to create it.

**Example**

*Table 19* illustrates the use of an import stub.

**Table 19. Use of import stub**

| Caller | Import stub | Callee |
|---|---|---|
| ...<br># link editor redirects this to .PLT1<br>call $r63=func<br>  | .PLT1:<br>ldw<br>$r63=.got.func[$r14]<br>mov $r9=$r63<br>;;<br>nop<br>;;<br>nop<br>;;<br>nop<br>;;<br>goto $r63<br>mov $r63=$r9<br>;;<br>  | func:<br>... |

The dynamic linker is allowed to implement *lazy binding*, whereby the global offset table entry created to hold the address of the called function is not resolved until the first time it is used in an import stub. Instead, the value of the global offset table entry is initialized by the dynamic linker to be the address of a secondary PLT entry that is also unique to the function being called. The secondary PLT entry transfers control to the dynamic linker's lazy binding entry point, which then resolves the reference, updates the GOT entry and completes the call.

In order for the implementation to perform lazy binding correctly, the application must conform to the following conventions for transfer of control to the dynamic linker's lazy binding entry point:

1. The link editor must reserve the first three 32-bit words of the global offset table. These words are initialized by the dynamic linker at program startup.
2. The relocation index for the function being called must be placed into R9, so that the dynamic linker can identify the target of the call. This value is an index into the portion of the dynamic relocation table addressed by the `DT_JMPREL` dynamic section entry. The designated relocation entry has type `R_ST200_JMP_SLOT`, and its offset specifies the global offset table entry referenced by the call.
3. A 4-byte identifier unique to the calling module must be placed into R10, so that the dynamic linker can identify the object from which the call originated, and thereby locate that object's relocation table. This identifier is found in the first 32-bit word of the global offset table.
4. The dynamic linker's lazy binding entry point is found in the second 32-bit word of the global offset table. (The third 32-bit word of the global offset table is reserved for future use.)

The link editor must create import stubs, secondary PLT entries, and allocate global offset table entries for any direct call that cannot be statically bound within the same dynamic object (including calls where a definition is present, but is not protected against pre-emption).

*Note: A dynamic linker that supports lazy binding should also provide a way of disabling it. Lazy binding may or may not be appropriate, depending on the operating environment. In a general purpose OS environment, it generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial call to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs, and the dynamic linker cannot resolve the symbol and terminates the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.*

**Example**

This is an example implementation of these conventions.

*Note: This example assumes that the global pointer value in* R14 *points exactly to the base of the global offset table. If* R14 *points to a non-zero constant offset from the base of the GOT, then the indexing offsets from* R14 *must be adjusted appropriately.*

**Figure 2. Sample procedure linkage table entries**

```
.PLT0:                          # initial special reserved entry in PLT

   ldw $r63=4[$r14]             # load lazy binding entry point
   mov $r11=$r63
   ;;
   ldw $r10=0[$r14]             # load module identifier
   ;;
   ...
   goto $r63                    # call lazy binding entry point
   mov $r63=$r11
   ;;

   ...
.PLT1:                          # import stub for symbol func

   ldw $r63=.got.func[$r14]     # load GOT entry for symbol func
   mov $r9=$r63
   ;;
   ...                          # call GOT entry for symbol func
   goto $r63
   mov $r63=$r9
   ;;

   ...
.PLT1a:                         # secondary PLT entry for symbol func

   goto .PLT0                   # load relocation index and branch to
   mov $r9=reloc_index          # common PLT0 entry
   ;;
```

The following steps show the procedure used to lazily resolve symbolic references.

1. When first creating the memory image of the program, the dynamic linker sets the first two entries of the global offset table. The first entry is initialized to contain the module identifier, and the second entry is initialized to point to the lazy binding function.
2. During execution, the program makes a call to the function `func`. The link editor has resolved this call to transfer control to `.PLT1`.
3. The code in `.PLT1` loads the GOT entry that contains the address of `func`, and then transfers control to it. The link editor has initialized this GOT entry so that it contains the address of the secondary PLT entry for `func`, `.PLT1a`, so control is actually passed to `.PLT1a`.
4. The code in `.PLT1a` loads the relocation index into R9, and transfers control to `.PLT0`. The relocation index is an index into the relocation table addressed by the `DT_JMPREL` dynamic section entry. The designated relocation entry has type `R_ST200_JMP_SLOT` and its offset specifies the GOT entry addressed by the code in `.PLT1`. The relocation entry also contains a symbol table index, which tells the dynamic linker the symbol that is being referenced, that is `func`.
5. `.PLT0` contains a PLT entry that is shared by all the secondary PLT entries (it is shared to reduce code size). It performs the call to the dynamic linker's lazy binding function. When `.PLT0` is reached, R9 contains the relocation index, R15-R23 contain the arguments to `func`, and R63 contains the program's return address.

The code in .PLT0 loads the values from the first two entries in the GOT. The address of the lazy binding function is loaded into R63, and the module identifier is loaded into R10. Then control is passed to the dynamic linker's lazy binding function.

*Note: At .PLT0,* R63 *contains the program's return address, so it is temporarily saved in* R11 *during the control transfer. Also note that the arguments to* `func` *are still in the parameter registers, so these cannot be used to hold the arguments for the dynamic linker.*

6. When the dynamic linker receives control, two scratch registers contain information it uses to relocate the function call: R9 contains the relocation index, and R10 contains the module identifier. The dynamic linker looks at the designated relocation entry, finds the symbol's value, stores this value in the GOT entry for `func`, and transfers control to `func`.
7. The GOT entry for `func` has been updated to contain the true address of `func`, so subsequent executions of .PLT1 transfer control directly to `func`, bypassing the call to the dynamic linker.

# 4 Coding examples

This chapter contains some example code sequences, and shows how the ST200 ELF file format represents their relocation.

*Note: In all of these examples, **imml** is used to represent the immediate extension, but depending on the instruction address alignment, the immediate extension may be by an **immr** following the instruction.*

## 4.1 Absolute data addressing

To place the absolute address of a data object in the data section, use the following:

```
.data4 var
```

This is represented as:

| Data | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0 | R_ST200_32 | var | 0 |

In an absolute executable, the link editor resolves this relocation. In a shared library, the absolute address of `var` is not known until load time, so the link editor creates a dynamic relocation, which instructs the dynamic loader to resolve it:

| Data | Dynamic relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0 | R_ST200_32 | var | 0 |

To access data by an absolute address in code, use the following:

```
ldw $r24=var
```

This is expanded to have an immediate extension:

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_HI23 | *x* | 0 |
| ldw $r24=0[$r0] | R_ST200_LO9 | *x* | 0 |

The relocations are resolved by the link editor.

If this code is in a main program executable and $x$ is defined in a dynamic library, then the link editor is not able to determine the absolute address of $x$. In this case, the link editor must define a symbol $x$ in the main program which pre-empts the definition of $x$ in the

dynamic library. If *x* is a function, the convention is to use the address of the import stub for *x*. If *x* is data, the convention is to create a complete copy of *x* in the `.dynbss` section. However, if the *x* defined in the dynamic library is data then it may have a static initializer, and this initial value must be transferred to the copy of *x* in the `.dynbss` section by the dynamic loader. The `R_ST200_COPY` relocation is used to instruct the dynamic linker to perform this copy.

| .dynbss entry | Dynamic relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `.space size, 0` | `R_ST200_COPY` | x | 0 |

The `R_ST200_COPY` is effectively a `memcpy`, where the source is definition of *x* in a shared library, the length is the size of that *x* as given by the symbol's `st_size` field, and the destination is given by the offset field of the relocation.

## 4.2 Addressing "own" data relative to gp

```
ldw $r24=@gprel(var)[$r14]
```

Generally, `@gprel(var)` does not fit in 9 bits, so this is expanded to have an immediate extension.

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_GPREL_HI23` | var | 0 |
| `ldw $r24=0[$r14]` | `R_ST200_GPREL_LO9` | var | 0 |

The `R_ST200_GPREL_xxx` relocations are resolved by the link editor.

## 4.3 Addressing external data in PIC code

```
ldw t1=@gotoff(var)[$r14]
;;
...
ldw $r24=[t1]
```

The first instruction is relocated. Generally, `@gotoff(var)` does not fit in 9 bits, so it is expanded to have an immediate extension.

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| `imml 0` | `R_ST200_GOTOFF_HI23` | var | 0 |
| `ldw t1=0[$r14]` | `R_ST200_GOTOFF_LO9` | var | 0 |

The R_ST200_GOTOFF_*xxx* relocations cause the link editor to create a global offset table entry for var (if one does not already exist) and to resolve these relocations to the *GP*-relative offset of this entry.

The global offset table entry is subject to the following dynamic relocation created by the link editor:

| Global offset table entry | Dynamic relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0 | R_ST200_32 | var | 0 |

At load time, the dynamic loader performs a full symbol lookup of var, and resolves the relocation to the run-time address of var.

As an optimization, if the link editor can determine that var is “own”, then it can avoid load-time symbol resolution as follows:

The GOT entry must contain the run-time address of var. The run-time address of a symbol is related to the link-time address of the symbol by the equation

```
S = @ltv(S) + BD
```

The R_ST200_REL32 performs the calculation *BD* + *A*. Because var is “own”, the link editor can determine the value @ltv(var), and it can use an R_ST200_REL32 relocation with the addend (*A*) set to @ltv(var) to calculate the run-time address of var the following is used:

| Global offset table entry | Dynamic relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0 | R_ST200_REL32 | null | @ltv(var) |

*Note: That if the link editor can determine that var is an “own” symbol, then these relocations can be optimized as described in Section 2.4.2: Relocation transformations on page 13.*

## 4.4 Materializing function pointers

To initialize a function pointer in the data segment, use the following:

```
label: .data4 func
```

This datum is relocated:

| Data | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0 | R_ST200_32 | func | 0 |

R_ST200_32 requires the absolute address of the function. This is handled in the same way as absolute data addressing (*Section 4.1 on page 26*).

To initialize a function pointer in code, the following is used:

```
ldw t1=@gotoff(func)[$r14]
```

This instruction is expanded to have an immediate extension, and relocated as follows:

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_GOTOFF_HI23 | func | 0 |
| ldw t1=0[$r14] | R_ST200_GOTOFF_LO9 | func | 0 |

The R_ST200_GOTOFF_*xxx* relocations cause the link editor to create a global offset table entry for var (if one does not already exist) and to resolve these relocations to the GP-relative offset of this entry.

The global offset table entry is subject to the following dynamic relocation created by the link editor:

| Global offset table entry | Dynamic relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0 | R_ST200_32 | var | 0 |

## 4.5 Direct procedure calls

```
mov $r1=$r14
call $r63=func
;;
mov $r14=$r1
```

The call instruction is relocated:

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| call $r63=0 | R_ST200_23_PCREL | func | 0 |

If func is not in the load module, or is pre-emptible, the R_ST200_23_PCREL relocation causes the link editor to create a import stub, secondary PLT entry and GOT entry for func, which is added to the text segment of the linked object. (Procedure linkage tables are described in more detail in *Section 3.3.3 on page 22*.) The global offset table entry is dynamically relocated using an R_ST200_JMP_SLOT relocation:

| Global offset table entry | Dynamic relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| .data4 0, 0 | R_ST200_JMP_SLOT | func | 0 |

The dynamic loader resolves the R_ST200_JMP_SLOT relocation using the address of func. If lazy binding is being performed, then this resolution happens the first time the function is called from this object, see *Section 3.3.3: Procedure linkage table on page 22*.

## 4.6 Materializing the GP value

The code sequence to materialize the GP value is:

```
   call $r63=.L1
   ;;
.L1:add $r14=$r63,@neggprel(.L1)
```

The add instruction is given an immediate extension, and relocated as follows:

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_NEGGPREL_HI23 | .L1 | 0 |
| add $r14=$r63,0 | R_ST200_NEGGPREL_LO9 | .L1 | 0 |

The link editor resolves these relocations.

If the addpc instruction is available on the target, then use the following code instead:

```
.L1:addpc $r14= @neggprel(.L1)
```

The addpc instruction is given an immediate extension, and relocated as follows:

| Instruction | Static relocation | | |
|---|---|---|---|
| | Type | Symbol | Offset |
| imml 0 | R_ST200_NEGGPREL_HI23 | .L1 | 0 |
| addpc $r14=0 | R_ST200_NEGGPREL_LO9 | .L1 | 0 |

The link editor resolves these relocations.

## 4.7 Use of the “link-time value” relocation

The `R_ST200_LTV32` relocation is used rarely. One use of it is in the Linux dynamic loader, when determining the base address displacement (*BD*) of the current shared object.

The Linux dynamic loader is unusual in that it must be position-independent (so that it can be loaded into any address space), and it must not have any dynamic relocations itself (because if it required to be dynamically relocated itself, that would require a recursive call of the dynamic loader).

So the dynamic loader must calculate *BD* without using any absolute addresses or dynamic relocations itself.

The link-time address of a symbol and the run-time address of a symbol are related by the following equation:

```
S = @ltv(S) + BD
```

so *BD* can be calculated as:

```
BD = S - @ltv(S)
```

*S* is the run-time address of the symbol, which for a code symbol can be obtained by adding a constant to the *PC*. `@ltv`(*S*) is the link-time address of the symbol, which can be calculated by the link editor and does not require any dynamic relocation.

The following code is used:

```
   .data
lab1:
   data4 @ltv(lab2) # this is relocated by R_ST200_LTV32

   .text
   ...
   lab2:
   ldw t1=@gprel(lab1)[$r14] # t1 now contains @ltv(lab2)
   add t2=$r14,@gprel(lab2) # t2 now contains lab2
   ;;
   ...
   sub $r16=$r63,t1 # lab2 - @ltv(lab2) gives BD
```

# 5 Acknowledgements

The ST200 series cores are based on technology jointly developed by Hewlett-Packard Laboratories and STMicroelectronics.

Linux® is a registered trademark of Linus Torvalds.

# 6 Revision history

**Table 20. Document revision history**

| Date | Revision | Changes |
|---|---|---|
| 24-Jun-2006 | A | Initial release. |
| 07-Nov-2006 | B | Moved to new template. No technical changes. |
| 04-Dec-2007 | C | Throughout: expressed register names in "`$r`" format rather than "`$r0`." .<br>Updated *Section 1.2: Additional documents on page 4*.<br>Added footnote to *Section 2.4.2: Relocation transformations on page 13*.<br>Added second example to *Section 4.6: Materializing the GP value on page 30*. |
| 24-Jun-2009 | D | Removed a reference to the ST220 from *Section 1.2: Additional documents on page 4*.<br>Added a footnote to *Table 5: Values for ELF_ST200_CORE on page 6* to identify cores no longer support by the current ST200 Micro Toolset. |
| 30-Aug-2012 | 5 | Updated document as the ST240 core is no longer supported:<br>– updated *Section 1.2: Additional documents on page 4*<br>– added footnote to `ELF_ST200_CORE_ST240` in *Table 5: Values for ELF_ST200_CORE on page 6* |