RM0245
Reference manual

ST200 run-time architecture

Overview

This document describes the common software conventions for the ST200 processor run-time architecture. It provides information needed to understand how the compiler behaves, for example, how it maps data structures into memory, and its alignment constraints. It will be of interest to anyone who has to design a system which shares data between an ST200 and another processor or who has to interpret memory dumps. It does not define operating system interfaces or any conventions specific to any single operating system.

Contents

Preface

This document is part of the ST200 documentation suite detailed below. Comments on this or other manuals in the ST200 documentation suite should be made by contacting your local STMicroelectronics sales office or distributor.

License information

The ST200 Micro Toolset is based on a number of open source packages. Details of the licenses that cover all these packages can be found on the CD in the file `license.htm`. This file is located in the `doc` subdirectory and can be accessed from `index.htm`.

ST200 documentation suite

The ST200 documentation suite comprises the following volumes:

ST200 Micro Toolset user manual (8063762)

This manual describes the ST200 Micro Toolset and provides an introduction to OS21. It covers the various cross tools and libraries that are provided in the toolset, the target platform libraries, how to boot OS21 applications from ROM and how to port applications which use STMicroelectronics' OS20 operating systems. Information is also given on how to build the open source packages that provide the compiler tools, base run-time libraries and debug tools and how to set up an ST Micro Connect.

ST200 Micro Toolset compiler manual (7508723)

This manual describes the software provided as part of the ST200 tools. It supports the development of ST200 applications for embedded systems. Applications can be developed in either a stand-alone environment, or under the OS21 real-time operating system. This manual also contains reference material relating to the ST200 Micro Toolset.

ST200 run-time architecture reference manual (7521848)

This manual describes the common software conventions for the ST200 processor run-time architecture.

OS21 user manual (7358306)

This manual describes the royalty free, light weight, OS21 multitasking operating system.

OS21 for ST200 user manual (7410372)

This manual describes the use of OS21 on ST200 platforms. It describes how specific ST200 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST200 platforms.

ST200 ELF specification (7932400)

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 is specific to the ST200 processor.

ST231 core and instruction set architecture reference manual (7645929)

This manual describes the architecture and the instruction set of the ST231 core as used by STMicroelectronics.

Terminology

The first ST Micro Connect product was named the “ST Micro Connect”; it is now known as the “ST Micro Connect 1” and the term “ST Micro Connect” is used to refer to the family of ST Micro Connect devices. The latest product is the “ST Micro Connect 2”. These names are abbreviated to “STMC”, “STMC1” and “STMC2”.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `sample code`, `keyboard input` and `file names`,
- *variables* and *`code variables`*,
- *`code comments`*,
- **screens**, **windows** and **dialog boxes,**
- **instructions**.

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,
- PIN NAMES and SIGNAL NAMES.

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1. Terminal strings of the language, that is, strings not built up by rules of the language, are printed in teletype font. For example, `void`.
2. Nonterminal strings of the language, that is, strings built up by rules of the language, are printed in italic teletype font. For example, *`name`*.
3. If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-`*`name`*.
4. Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’).
5. Alternatives are separated by vertical bars (‘`|`’).
6. Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’).
7. Items which may be repeated appear in braces (‘`{`’ and ‘`}`’).

Acknowledgements

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

Microsoft® is a registered trademark of Microsoft Corporation in the United States and/or other countries.

1 Introduction

The run-time architecture defines many, but not all, of the conventions necessary to compile, link, and execute a program on an operating system that supports these conventions. Its purpose is to ensure that object modules produced by different compilers can be linked together into a single application, and to specify the interfaces between the compilers and the linker, and between the linker and operating system.

The run-time architecture does not specify the Application Programming Interface (API), the set of services provided by the operating system to the program, or certain conventions that are specific to each operating system. Thus, conformance to the run-time architecture alone is not sufficient to produce a program that executes on all ST200 processor platforms. However, it does allow many of the development tools to be shared among various operating systems.

When combined with the instruction set architecture, an API, and system-specific conventions, this run-time architecture leads to an Application Binary Interface (ABI). In other words, an ABI is the composition of an API, system-specific conventions, a hardware description, and a run-time architecture.

1.1 Objectives of the run-time architecture

This document defines the software interfaces needed to ensure that the ST200 software operates correctly together. The intent is to define as small a set of interface specifications as possible, while still meeting the following goals:

- support 32-bit addressing and data types
- provide high performance
- ease porting
- ease implementation and use
- be suffiently complete to ensure software compatibility

The aim is to provide sufficiently complete interfaces between the different software products that they can be provided by different independent software vendors (ISV)s and still work together. These include compilers, linkers, simulators, applications, and dynamic link libraries. The goal is to have one standard, so software is portable on ST200 systems.

1.2 About the conventions

ANSI C serves as the reference programming language. By defining the implementation of C data types, the software conventions can give precise system interface information without resorting to assembly language. Giving C language bindings for system services does not preclude bindings for other programming languages. Moreover, the examples given here are not intended to specify the C language available on the system.

1.3 Glossary

The following terms are used in this document:

ABI Application binary interface, through which the application gains access to the operating system.

API Application programming interface, the calling conventions through which the application gains access to the operating system.

Absolute address In this document, the term absolute address refers to a virtual address, not a physical address. It is an address within the process' address space that is computed as an absolute number, without the use of a base register.

Binding The process of resolving a symbolic reference in one module by finding the definition of the symbol in another module, and substituting the address of the definition in place of the symbolic reference. The linker binds relocatable object modules together, and the dynamic loader binds executable load modules. When searching for a definition, the linker and dynamic loader search each module in a certain order, so that a definition of a symbol in one module has precedence over a definition of the same symbol in a later module. This order is called binding order.

Dynamic link library (DLL)

A library that is prepared by the linker for quick loading and binding when a program is invoked, or while the program is running. A DLL is designed so that its code is shared by all processes that are bound to it. (Also called **shared library**, **dynamic library**.)

Dynamically shared object (DSO)

See *Load module*.

Execution time The time during which a program is actually executing, not including the time during which it and its DLLs are being loaded.

Function pointer A reference or pointer to a function.

Global offset table (GOT)

A table in a position-independent load module that contains the absolute addresses of objects referenced by the code in the module. (Also called **linkage table**.)

Global pointer (GP) The global pointer value is an address at a constant offset to the base of the GOT and is selected by the link editor.

Internal procedure A procedure that can only be called from another procedure in the same load module.

ISA Instruction set architecture.

Link register (LR) The architectural link register used by the `call` and `return` mechanism.

Link time The time when a program or DLL is processed by the linker. Any activity taking place at link time is static.

Linkage table See *Global offset table (GOT)*.

Load module An executable unit produced by the linker, either a main program or a DLL. A program consists of at least a main program, and may also require one or more DLLs to be loaded to satisfy its dependencies. (Also called **dynamically shared object**.)

Millicode Specialized program code subroutines that are used to add more complex functionality to a base instruction set. These subroutines are usually designed with a streamlined calling convention for efficiency.

Own data Data belonging to a load module that is referenced directly from that load module and that is not subject to the binding order. If a module references a data item symbolically, and another module earlier in the binding order defines an item with the same symbolic name, the reference is bound to the data item in the earlier module. If this is the case, the data is not “own”. Typically, own data is local in scope.

PC-relative addressing

Code that uses its own address (commonly called the program counter, or “PC”) as a base register for addressing other code and data.

Position-independent code (PIC)

This term has a dual meaning. First, position-independent code is designed so that it contains no dependency on its own load address; usually, this is accomplished by using PC-relative addressing so that the code does not contain any absolute addresses. Second, it implies that the code is designed for dynamic binding to global data; this is usually done by using indirect addressing through the global offset table.

Preserved register A register that is guaranteed to be preserved across a procedure call.

Procedure linkage table (PLT)

If a load module calls a function in another load module, it does so using a piece of stub code that redirects the call. These stubs are collected together in the procedure linkage table.

Program invocation time

The time when a program or DLL is loaded into memory in preparation for execution. Activities taking place at program invocation time are generally performed by the system loader or dynamic loader.

Protection area A portion of a segment that shares common access protection.

Scratch register A register that is not preserved across a procedure call.

Section A part of an object file that holds information for linking. Object file sections contain program instructions, data, symbol table, relocations, debug tables and so on.

Segment An area of memory that has specific attributes, and behaves as a fixed unit at run-time. All items within a segment have a fixed address relationship to one another at execution time, and have a common set of attributes. For example, the program text segment is defined to contain the main program code, and may be assumed to be readable, executable, and not writable.

Static (1) Any data or code object that is allocated at a fixed location in memory and whose lifetime is that of the entire process, regardless of scope; (2) A binding that takes place at link time rather than program invocation or execution time.

Stack Pointer (SP) Memory stack pointer.

Thread local storage (TLS)

A mechanism by which variables are allocated such that there is one instance of the variable per extant thread. (Also known as **thread-private data**.)

Thread pointer (TP) Pointer to the current thread's local state.

TLB Translation lookaside buffer which provides memory translation and protection.

1.4 Bibliography

A number of references are made to "K&R" within this document, they refer to "*The C Programming Language*" by Kernighan and Ritchie.

2 ST200 architecture

It is assumed that applications conforming to this specification run in a software environment provided by some operating system, and that additional conventions are specified as part of the Application Binary Interface (ABI) for that operating system.

Programs intended to execute directly on an ST231 processor use the instruction set, instruction encoding, and instruction semantics defined in the *ST231 core and instruction set architecture reference manual* (7645929).

Three points deserve explicit mention.

- A program may assume all documented instructions exist.
- A program may assume all documented instructions work.
- A program may use only instructions defined by the architecture.

In other words, from a program's perspective, the execution environment provides a complete and working implementation of ST200.

This does not imply that the underlying implementation provides all instructions in hardware, only that the instructions perform the specified operations and produce the specified results. The software conventions neither place performance constraints on systems nor specify what instructions must be implemented in hardware. A software emulation of the architecture could conform to these conventions.

These conventions are intended for application use, and so use only features found in user mode. Applications should assume that they execute in user mode, and that any attempt to use processor resources restricted to system mode causes a trap that may terminate the process.

Application use of the **sbrk** operation is subject to the following conventions:

- Immediate operand value 0 is reserved for debugger breakpoints.
- Immediate operand value 1 indicates that a fatal run-time error has been detected.

Note: The immediate operand of an sbrk operation is not saved in any architectural state. Software must therefore decode the instruction to obtain this operand.

2.1 Addressing and protection

The features of the processor architecture that are described in the *Memory translation and protection* chapter of the *ST231 core and instruction set architecture reference manual* (7645929) are intended for the exclusive use of the operating system software, with the following exception:

The operating system software may provide access to certain page attributes, including caching and ordering attributes, through its API. The use of such features is ABI specific.

2.2 Interruptions

The features of the processor architecture that are described in the *Traps (exceptions and interrupts)* chapter of the *ST231 core and instruction set architecture reference manual* (7645929) are intended for the exclusive use of the operating system software. Interruptions include Faults, Traps, External Interrupts, and Aborts.

3 Memory model

A memory model is the organization of memory segments and protection areas that an application process uses.

These conventions define a memory system with a flat 32-bit address space. On a system that uses virtual memory, each process may run its own flat 32-bit virtual address space. On systems without virtual memory there is a single flat 32-bit physical address space.

Any operating system ported to the ST200 may divide this address space into different portions, and assign specific uses to each portion. This chapter describes the types of memory segments and protection areas that an application process uses, and documents the assumptions that an application may make about those segments. From a different perspective, it documents the minimum requirements that must be satisfied by an operating system with respect to its allocation of these program segments in the virtual address space.

The term “segment” is used here to identify an area of memory that has specific uses within an application and has no fixed address relationship to any other segment. Thus, relative distances between any two items belonging to the same segment are constant once the program has been linked, but the distance between two items in different segments is not fixed. It does not imply the use of hardware segmentation.

Segments are composed of one or more protection areas. The term “protection area” is used to indicate an area of memory that has common protection attributes.

3.1 Program segments

Table 1 lists the types of program segments that are defined by the run-time architecture, and defines the minimum set of attributes that an operating system must provide for these segments.

The shareable attribute indicates whether or not the memory contained within such a segment may be shared between two or more processes. For text segments, this implies that an OS will probably not grant write access, in order to make the text segment pure. For this reason, the run-time architecture does not place anything into the text segment that may need to be written at either program invocation time or execution time.

Table 1. Program segment attributes

Segment type	Shareable	Quantity	Address by	Contents
Text	Yes	1 or more for each load module	Program counter PC, absolute address, global offset table (GOT), global pointer (GP)	Text, unwind information, constants and literals
Data	No	Any	Absolute address, GOT, GP, PC	Data, bss
Heap	No	Any	Pointer	Heap data
Stack	No	1	Stack pointer (SP)	Memory stack

The run-time architecture does not specify how an OS will make a particular segment shareable.

A program consists of several load modules: the main program, and one for each dynamic link library (DLL) that it uses. Each load module is compiled and linked in a particular object code model.

Two main object code models are supported.

Absolute code Instructions can hold absolute addresses under this model. To execute properly, absolute code must be loaded at a specific address (on systems that use virtual memory, this means a specific virtual address).

Position-independent code Under this model, instructions must hold relative addresses, not absolute addresses. Consequently, the code is not tied to a specific load address, it will execute properly at various positions in memory. However, the relative offsets between the text and data segments are fixed, so no text or data segment may be moved relative to another text or data segment. In particular, global data may not be moved relative to code.

The main program load module may be either absolute or position-independent. A DLL must be position-independent, to allow text sharing. DLL text segments can be loaded at various addresses without having to change the segment images. Thus multiple processes can share a single DLL text segment, even though the segment resides at a different virtual address in each process.

Each load module consists of at least one text segment and one data segment. In an **absolute** load module, the addresses of these segments are fixed at link time, in a **position-independent** load module, the addresses of the segments are not fixed at link time. If the address of a segment is not fixed at link time, that segment may not be accessed by absolute address, it must instead be accessed PC-relative, GP-relative, or indirectly using the global offset table. The GP register and its conventions are described in *Chapter 7: Procedure linkage on page 27*.

Each OS is expected to provide some form of heap management, although the run-time architecture does not have any explicit dependencies on such. However, the API for obtaining heap memory is OS dependent, and the run-time architecture places no restrictions on the locations or contiguity of separately-allocated items from the heap.

3.2 Data allocation

Data may be allocated in the text segment, data segment or stack as indicated by *Table 1: Program segment attributes on page 14*. This section gives the rules for how the data is accessed in these various locations.

3.2.1 Global variables

In absolute code, access to a global that is known to be defined in the same load module can be made with an absolute address.

In position-independent code, access to a global that is known to be defined in the same load module or to a static local can be made with a GP-relative offset.

Access to a global variable that is not known (at compile time) to be defined in the same load module must be indirect. Each load module has a global offset table in its data segment, pointed to by the GP value; code must load a pointer to the global variable from the global offset table, then access the global variable through the pointer.

3.2.2 Local static data

In absolute code, access to local static data can be made with an absolute address.

In position-independent code access to local static data can be made with a GP-relative offset.

3.2.3 Constants and literals

Constants and literals may be placed in the text segment or in the data segment. They may be accessed in the same way as local static data.

In position-independent code, absolute address constants must not be placed in the text segment.

3.2.4 Local memory stack variables

Access is SP-relative. Stack frames must always be aligned on a 8-byte boundary. That is, the stack pointer register must always be aligned on a 8-byte boundary.

4 Data representation

This chapter describes the size, alignment requirements, and hardware representation of the standard C data types.

Applications running in a 32-bit environment use the “ILP32” data model: integers, long, and pointers are 32 bits. Within this specification, the term **halfword** refers to a 16-bit object, the term **word** refers to a 32-bit object, the term **doubleword** refers to a 64-bit object, and the term **quadword** refers to a 128-bit object.

4.1 Fundamental types

Table 2 lists the scalar data types supported by the architecture. Sizes and alignments are shown in bytes.

Table 2. Scalar types

Type	C	Size	Alignment	Hardware representation
Integral	char signed char	1	1	signed byte
	unsigned char	1	1	unsigned byte
	short signed short	2	2	signed halfword
	unsigned short	2	2	unsigned halfword
	int signed int long signed long enum	4	4	signed word
	unsigned int unsigned long	4	4	unsigned word
	long long signed long long	8	8	signed doubleword
	unsigned long long	8	8	unsigned doubleword
Pointer	*any-type** *any-type*(*)()	4	4	unsigned word
Floating-point	float	4	4	IEEE single precision
	double	8	8	IEEE double precision
Complex	float_complex	8	4	IEEE single precision pair, real part first
	double_complex	16	8	IEEE double precision pair, real part first

A null pointer (for all types) has the value zero.

Note: Enumerated types are considered signed only if one or more of the enumeration constants defined for that type are negative. If all enumeration constants are non-negative, the type is considered unsigned.

4.2 Aggregate types

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, is always a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The content of any padding is undefined.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member; an array object is aligned on the same boundary as its element type.
- Each structure member is assigned to the lowest available offset with the appropriate alignment. This may require **internal padding**, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require **tail padding**, depending on the last member.

In the following figures, members' byte offsets appear in the upper right corners for little endian, in the upper left corners for big endian.

Figure 1. Structure smaller than a word

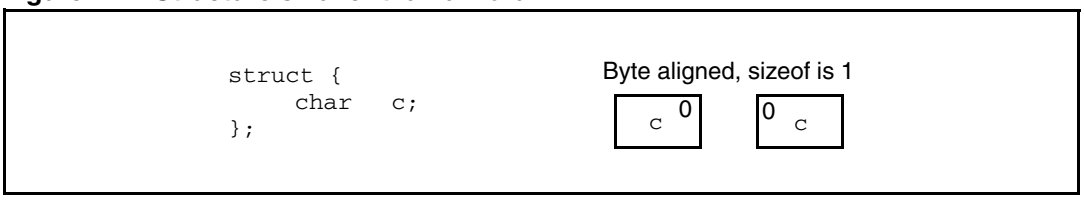


Figure 2. No padding

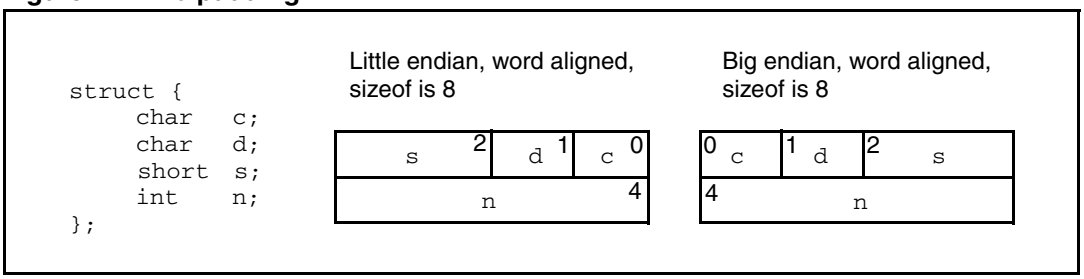


Figure 3. Internal padding

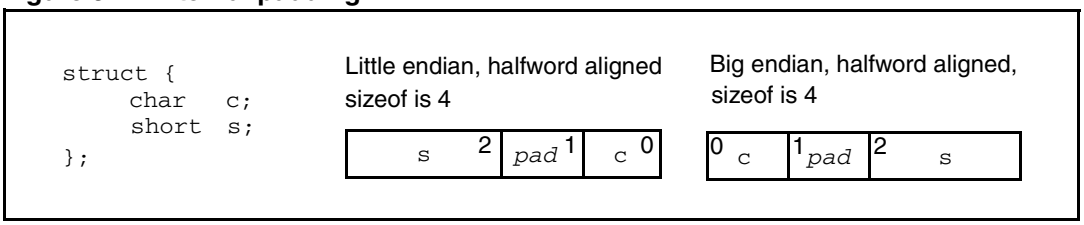


Figure 4. Internal and tail padding

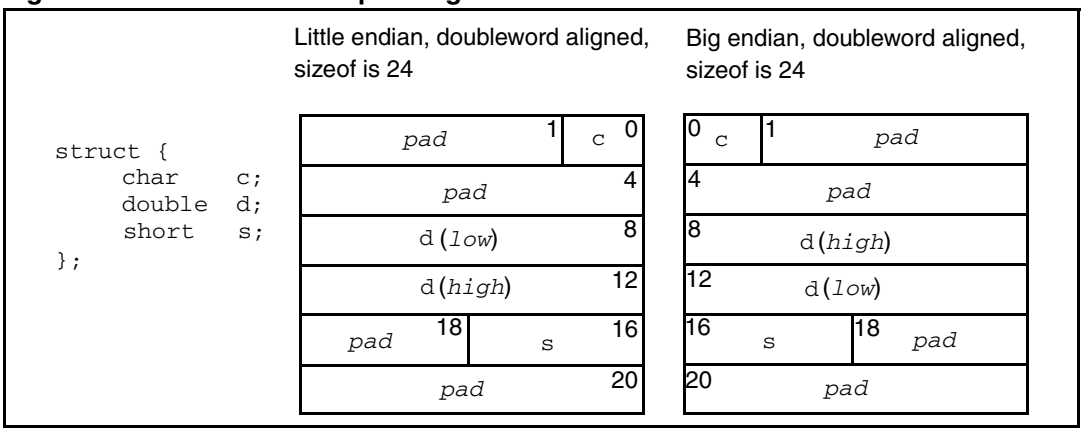
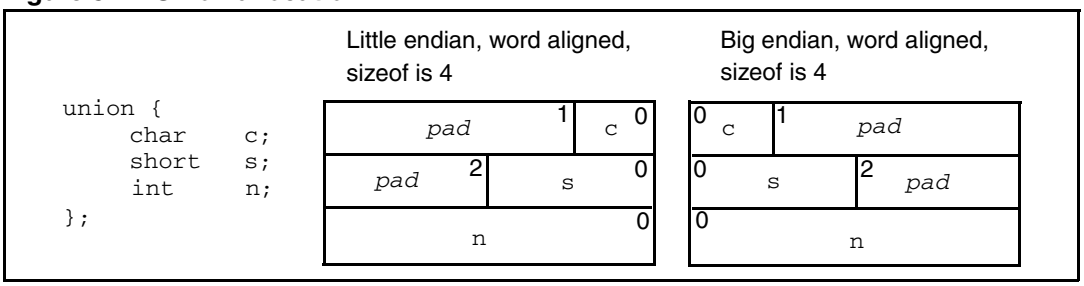


Figure 5. Union allocation



4.3 Bit-fields

C struct and union definitions may have **bit-fields** that define integral objects with a specified number of bits.

The ST200 run-time architecture does not specify the signedness of bit-field types. This is specified by the ISO C language standard for all bit-field types except for plain int. The signedness of plain int bit-fields should be specified by the compiler implementation.

Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions.

- Bit-fields are allocated least significant bit (LSB) to most significant bit (MSB) (from right to left) for little endian. They are allocated MSB to LSB (left to right) for big endian.
- A bit-field must entirely reside in a storage unit appropriate for its declared type. For example, a bit-field of type short must never cross a halfword boundary.
- Bit-fields may share a storage unit with other struct or union members, including members that are not bit-fields. Of course, each struct member occupies a different part of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union. Zero-length unnamed bit-fields force the alignment of subsequent members to the boundary corresponding to the size of the bit-field's type.

The *Figure 6* to *Figure 8* show struct and union member byte offsets in the upper corners; bit numbers appear in the lower corners.

Figure 6. Bit numbering

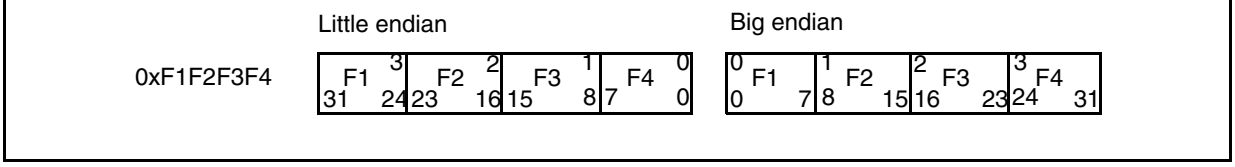


Figure 7. Union allocation

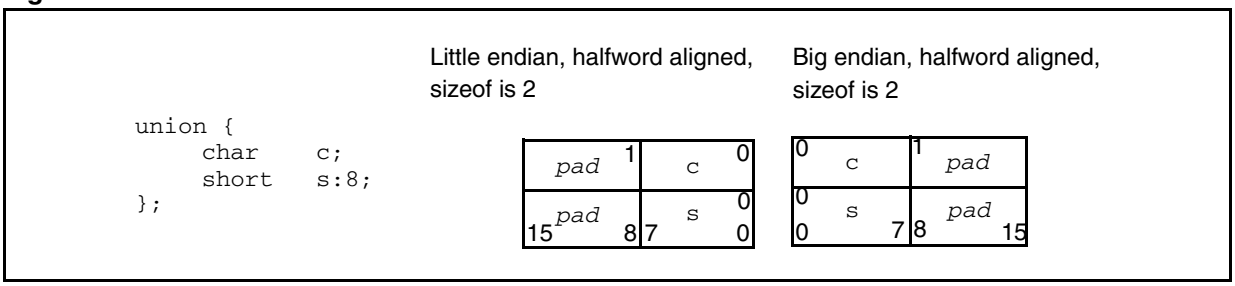
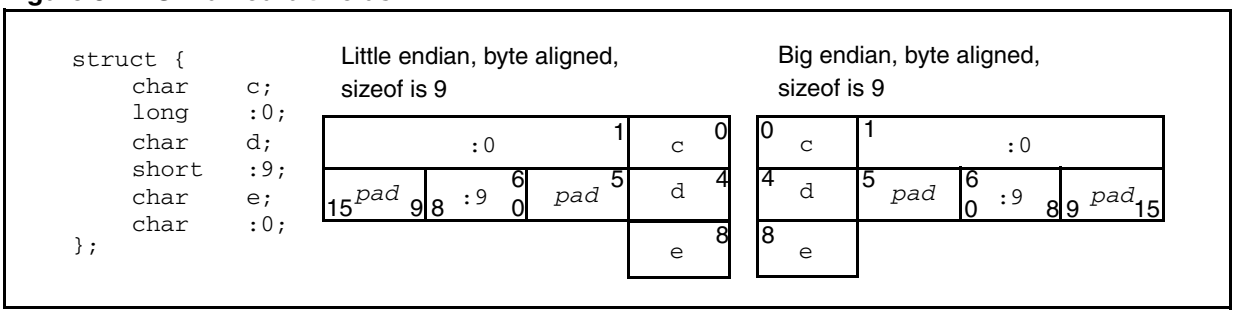


Figure 8. Unnamed bit fields



Note: Unnamed bit-fields do not affect the alignment of the structure.

As the examples show, `int` and `long` bit-fields (including signed and unsigned) usually pack more densely than smaller types. One can use `char` and `short` bit-fields to force allocation within those types, but `int` is generally more efficient.

5 Register usage

This chapter describes the rules that govern how ST200 registers are used by an application.

5.1 Partitioning

Registers are partitioned into the following classes:

- scratch registers may be destroyed by a procedure call; the caller must save these registers before a call if needed (caller-saves),
- preserved registers must not be destroyed by a procedure call; the callee must save and restore these registers if used (callee-saves),
- constant or read-only registers contain a fixed value that cannot be changed by the program,
- special registers are used in the call/return mechanism. The conventions for these registers are described individually *Section 5.2*.

5.2 General registers

General registers are used for integer arithmetic and other general-purpose computations. *Table 3* lists the general registers.

Table 3. General registers

Register	Class	Usage
R0	constant	always zero
R1-R7	preserved	
R8-R11	scratch	
R12	special	memory stack pointer (SP)
R13	special	reserved as a thread pointer (TP)
R14	preserved	global pointer (GP)
R15	scratch	struct/union return pointer register
R16-R23	scratch	procedure argument/return value
R24-R62	scratch	
R63	special	link register (Also known as LR)

R0 Wired to the value 0.

R1-R7 These preserved registers have no specific role in the function calling sequence. A function must preserve their values for the caller (callee-saves).

R8-R11	These scratch registers have no specific role in the function calling sequence. Functions do not have to preserve their values for the caller (caller-saves).
R12	The stack pointer holds the limit of the current stack frame, which is the address of the stack's bottom-most, valid word. At all times, the stack pointer must point to a 0 mod 8 aligned area. The stack pointer is also used to access any memory arguments upon entry to a function. Except in the case of dynamic stack allocation (for example, `alloca`), this register is preserved across any functions called by the current function. The compiler can assume that a function call will preserve the stack pointer. A call to a function that does not preserve the stack pointer must be indicated to the compiler (for example, by a pragma) to ensure the compiler generates code that behaves properly. Failure to notify the compiler leads to undefined behavior. The standard function calling sequence does not include any method to detect such failures. This allows the compiler to use the stack pointer to reference stack items without having to set up a frame pointer for this purpose.
R13	This register is reserved for use as a thread pointer. The usage of this register is ABI specific. Programs conforming to these conventions may not modify this register.
R14	The global data pointer (GP) register. In position-independent code, this register is designated to hold the address of the currently addressable global data segment. At any procedure call from position-independent code, R14 must contain the GP value for the caller. This guarantees that an import stub (see *Section 10.2: Import stubs on page 46*) can access the global offset table. It also guarantees that the GP value is already correctly set on entry to a position-independent internal procedure. R14 is a preserved register, therefore its value is preserved by all procedure calls. In general, if a procedure needs the GP value, it must establish it before first use. However, a position-independent internal procedure can assume that on entry R14 already contains the correct GP value, and omit the code to establish the GP value. In absolute code, and in position-independent leaf functions, R14 has no special role, and is available for general use. However, it must still be treated as a preserved register.
R15	The struct/union return pointer register. If the function called returns a struct or union value larger than 32 bytes, then register R15 contains, on entry, the appropriately aligned address of the caller-allocated area to contain the value being returned. This register is not guaranteed to be preserved by the called procedure (caller-saves).

R16-R23	Argument values up to 32 bytes are passed in these registers. Arguments beyond these registers appear in memory, as explained in *Section 7.4: Parameter passing on page 30* (this section also contains a discussion on structure and union arguments). Within the called function, these registers are local scratch registers and are not preserved for the caller. Return values up to 32 bytes also appear in these registers.
R24-R62	These scratch registers have no specific role in the function calling sequence. Functions do not have to preserve their values for the caller (caller-saves).
R63	This register contains the return address on entry to a procedure. It is implicitly read by the indirect **goto** and **call** instructions that are used for indirect local jumps and indirect function calls. This register is also known as the link register (LR).

5.3 Branch registers

Branch registers are single-bit-wide registers used for controlling conditional branches. *Table 4* lists the branch registers.

Table 4. Branch registers

Register	Class	Usage
B0-B7	scratch	These scratch registers have no specified role in the function calling sequence. Functions do not have to preserve their value for the caller (caller-saves).

6 Memory stack

This chapter describes the organization of the memory stack of procedure frames.

The memory stack is used for local dynamic storage, spilled registers, and parameter passing. It is organized as a stack of procedure frames, beginning with the main program's frame at the base of the stack, and continuing towards the top of the stack with nested procedure calls. At the top of the stack is the frame for the currently active procedure. (There may be some system-dependent frames at the base of the stack, prior to the main program's frame, but an application program may not make any assumptions about them.)

The memory stack begins at an address determined by the operating system, and grows towards lower addresses in memory. The stack pointer register, SP, always points to the lowest address in the current, topmost, frame on the stack.

Each procedure creates its frame on entry by subtracting its frame size from the stack pointer, and removes its frame from the stack on exit by restoring the previous value of SP (usually by adding its frame size, but a procedure may save the original value of SP when its frame size may vary).

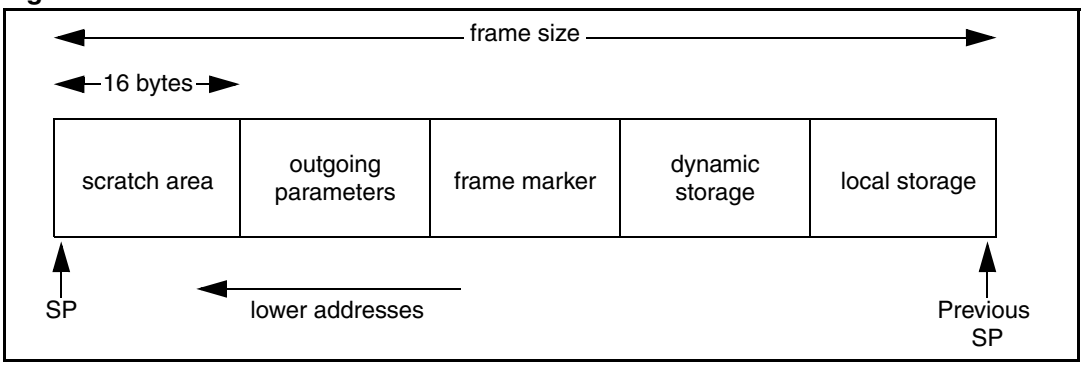
Not every procedure needs a memory stack frame. However, every non-leaf procedure needs to save at least its return link, so there is an activation record for every non-leaf procedure on the memory stack.

6.1 Procedure frames

A procedure frame consists of five regions, as illustrated in *Figure 9*.

- Local storage. A procedure may store local variables, temporaries, and spilled registers in this region.
- Dynamically allocated stack storage. This is a variable-sized region (initially zero length) that can be created by the C library `alloca` routine and similar routines.
- Frame marker. This optional region may contain information required for unwinding through the stack (for example, a copy of the previous stack pointer).
- Outgoing parameters. Parameters in excess to those passed in registers are stored in this area of the stack frame. A procedure accesses its incoming parameters in the outgoing parameter region of its caller's stack frame.
- Scratch area. This 16-byte region is provided as scratch storage for procedures that are called by the current procedure. Leaf procedures do not need to allocate this region. A procedure may use the 16 bytes at the top of its own frame as scratch memory, but the contents of this area may be destroyed by a procedure call.

Figure 9. Procedure frame



The stack pointer must always be aligned at an 8-byte boundary. This implies that all stack frames must be a multiple of 8 bytes in size.

An application may not write to memory below the stack pointer. Any memory below the stack pointer may be destroyed at any time.

Most procedures are expected to have a fixed size frame, and the conventions are biased in favor of this. A procedure with a fixed size frame may reference all regions of the frame with a compile-time constant offset relative to the stack pointer. Compilers should determine the total size required for each region, and pad the local storage area to make the total frame size a multiple of 8 bytes. The procedure may then create the frame by subtracting an immediate constant from the stack pointer in the prologue, and remove the frame by adding the same immediate constant to the stack pointer in the epilogue.

If a procedure has a variable-size frame (for example, it contains a call to `alloca`), it should make a copy of SP to serve as a frame pointer before subtracting the initial frame size from the stack pointer. It may then restore the previous value of the stack pointer in the epilogue without regard for how much dynamic storage has been allocated within the frame. It may also use the frame pointer to access the local storage region, since offsets from SP will vary.

However, a frame pointer, as described above, is not required provided that the compiler uses an equivalent method of addressing the local storage region correctly before and after dynamic allocation, and provided that the code satisfies conditions imposed by the stack unwinding mechanism.

To expand the stack frame dynamically, the scratch area, outgoing parameters, and frame marker regions, which are always located relative to the current stack pointer, must be relocated to the new top of the stack. If the scratch area and outgoing parameter area are both clear of any live values, there is no actual work involved in relocating these areas. For procedures with dynamically sized frames, it is recommended that the previous stack pointer value be stored in a preserved general register instead of the frame marker, so that the frame marker is also empty. If the previous stack pointer is stored in the frame marker, the code must take care to ensure that the stack is always unwindable while the stack is being expanded.

Other issues depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses any stack frame region beyond the purpose described here. For example, the outgoing parameter region may be used as scratch storage whenever it is not needed for passing parameters.

6.2 Rationale

The 16-byte scratch region is provided so that a procedure can use stack storage immediately on entry without having to wait a cycle until the stack pointer has been decremented and its own frame is established.

7 Procedure linkage

This chapter discusses the linkage conventions and the details of the procedure calling sequence, including parameter passing and return values.

7.1 External naming conventions

The standard naming convention, referred to as the “C” convention, specifies that all external symbols have linkage names identical to the source language identifier. There are no leading or trailing underscores. Other languages may establish other conventions, but they should provide a mechanism to define and reference symbols with “C” linkage.

7.2 Types of calls

The following types of procedure calls are defined:

- **Direct calls**. Direct calls within the same load module may be made directly to the entry point of the target procedure.
- **Direct dynamically linked calls**. These calls are normally routed through an import stub, but if the call is known to be to a DLL, or the compiler deduces through some means that there is a strong probability that the call is to a DLL, then the import stub can be inlined at compile time. The import stub obtains the address of the procedure entry point from the global offset table. Although coded in source as a direct call, dynamically linked calls become indirect.
- **Indirect calls**. A function pointer must point to the address of the function entry point for the target function. The compiler must generate code for an indirect call.
- **Special calls**. Other special calling conventions are allowed to the extent that the compiler and the run-time library agree on the convention, and provided that the stack may be unwound through such a call. Such calls are outside the scope of this document. See *Section 7.6 on page 34* for a discussion of stack unwind requirements.

7.3 Calling sequence

Direct and indirect procedure calls are described in the following sections.

7.3.1 Direct calls

Direct procedure calls follow the sequence of steps shown in *Figure 10 on page 29*. The following paragraphs describe these steps in detail.

1. **Preparation for call**. Values in scratch registers that must be kept live across the call must be saved. They can be saved by either copying them into preserved registers or by saving them on the memory stack.

   The parameters must be set up in registers and memory as described in *Section 7.4*.
2. **Procedure call**. All direct calls are made with a call immediate instruction, which writes the link register (also known as LR) for the return link.

   For direct local calls the PC-relative displacement to the target is computed at link time. Compilers may assume that the standard displacement field in the **call** instruction is

sufficiently wide to reach the target of the call. If the displacement is too large, the linker must supply a branch stub at some convenient point in the code; compilers must guarantee the existence of such a point by ensuring that code sections in the relocatable object files are no larger than the maximum reach of the **call** instruction. With a 23-bit displacement and word addresses, the maximum reach is 16 Mbytes in either direction from the point of call.

Direct calls to other load modules cannot be statically bound at link time, so the linker must supply an import stub for the target procedure; the import stub obtains the address of the target procedure from the global offset table. The **call** instruction can then be statically bound using the PC-relative displacement to the import stub.

The **call** instruction saves the return link in the link register, which is aliased to general register R63.

3. **Import stub (direct external calls only)**. The import stub is allocated in the load module of the caller, so that the call instruction may be statically bound to the address of the import stub. The import stub obtains the address of the target procedure's entry point from the global offset table. In position-independent code (PIC), it must access the global offset table using the current GP (which means that the GP must be valid at the point of call). In absolute code, it can access the global offset table using an absolute reference, so the GP does not need to be valid at the point of call. The import stub then branches to the target entry point.

   The detailed operation of an import stub is ABI specific.

   When the target of a call is in the same load module, an import stub is not used. However, for position-independent code, the GP value must still be valid for the caller at the point of call, so that if the target is an internal function, it can assume that the GP value is already correctly set.

   The compiler may choose to generate calling code that performs the functions of the import stub. This saves a branch compared to using the import stub, but is less efficient than a direct call within the same load module. Therefore, the compiler should only do this if it deduces that call target is in a separate load module, or that there is a high probability of this.

4. **Procedure entry**. The prologue code in the target procedure is responsible for allocating a frame on the memory stack, if necessary. It may use the 16 bytes at the top of its caller's memory stack frame as scratch area.

   If it is a non-leaf procedure, it must save the return link in the memory stack frame.

   The prologue must also save any preserved registers that will be used in this procedure.

   If it is a position-independent procedure that makes calls or accesses global data, then it must establish the GP value in the GP register. The GP register (R14) is a preserved register, so it must be saved before being modified. A position-independent internal function may assume that the GP register already contains the correct value.

   A position-independent leaf procedure that accesses global data is not required to put the GP value in R14, it may use a scratch register instead, thus avoiding the need to save and restore R14.

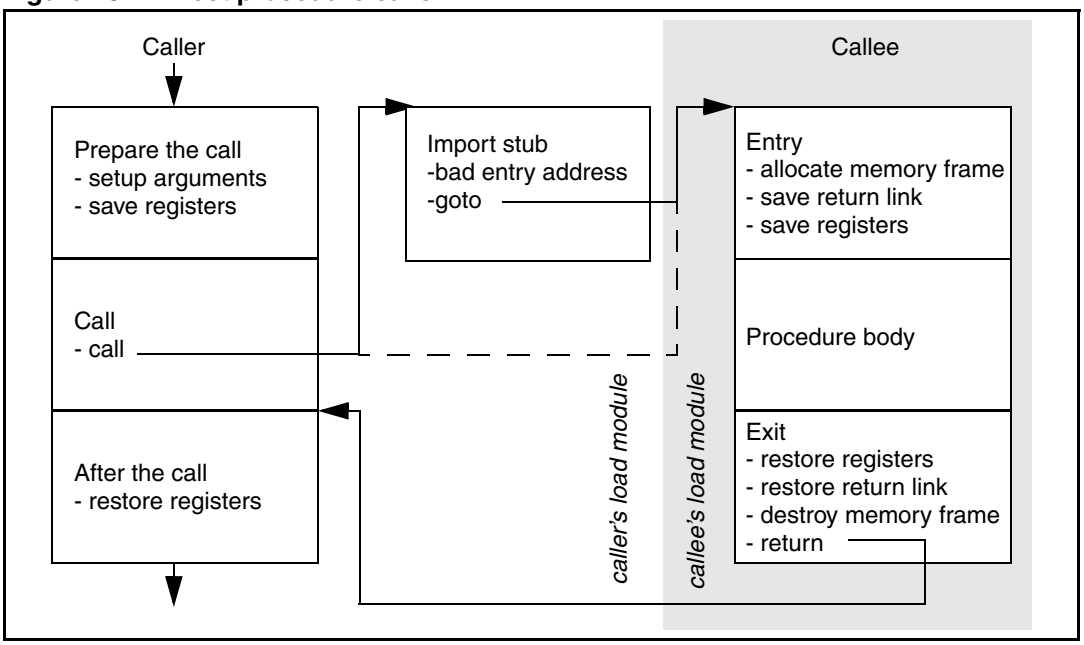
5. **Procedure exit**. The epilogue code is responsible for restoring the return link and any preserved registers that were saved.

   If a memory stack frame was allocated, the epilogue code must deallocate it.

   Finally, the procedure exits by branching through the link register with the **return** instruction.

6. **After the call**. Any saved values should be restored.

Figure 10. Direct procedure calls



7.3.2 Indirect calls

Indirect procedure calls follow nearly the same sequence, except that the branch target is established indirectly. This sequence is illustrated in *Figure 11*.

1. **Preparation for call**. Indirect calls are made by loading the entry point address into the link register.

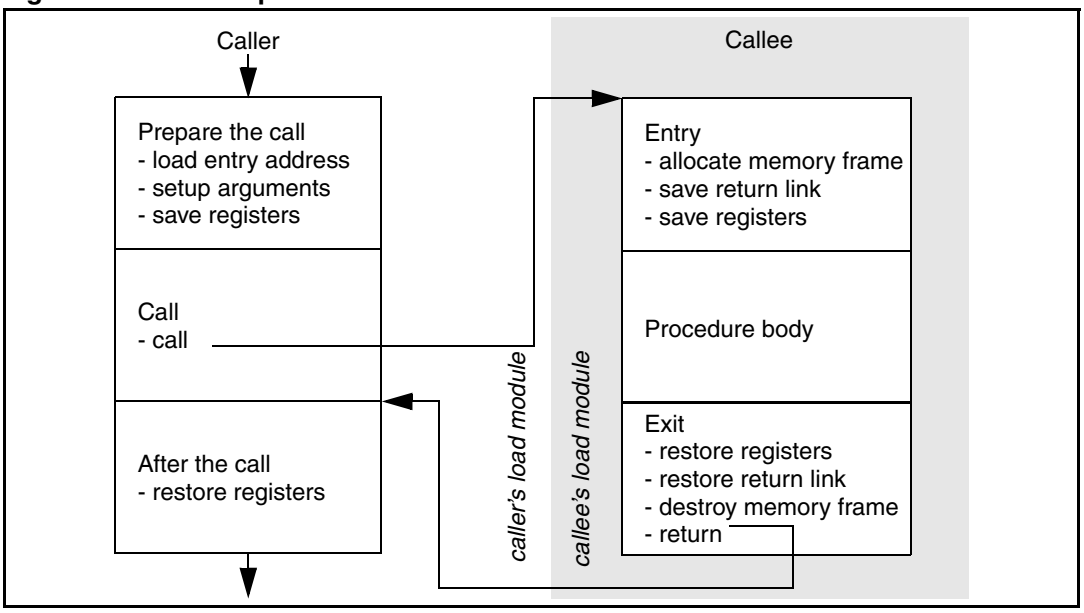
   Values in scratch registers that must be kept live across the call must be saved. They can be saved by either copying them into preserved registers or by saving them on the memory stack. The parameters must be set up in registers and memory as described in *Section 7.4*.

2. **Procedure call**. All indirect calls are made with the **call** indirect instruction, which reads and writes the link register.

   The **call** instruction saves the return link in the link register.

3. **Procedure entry, exit, and return**. The remainder of the calling sequence is the same as for direct calls.

Figure 11. Indirect procedure calls



7.4 Parameter passing

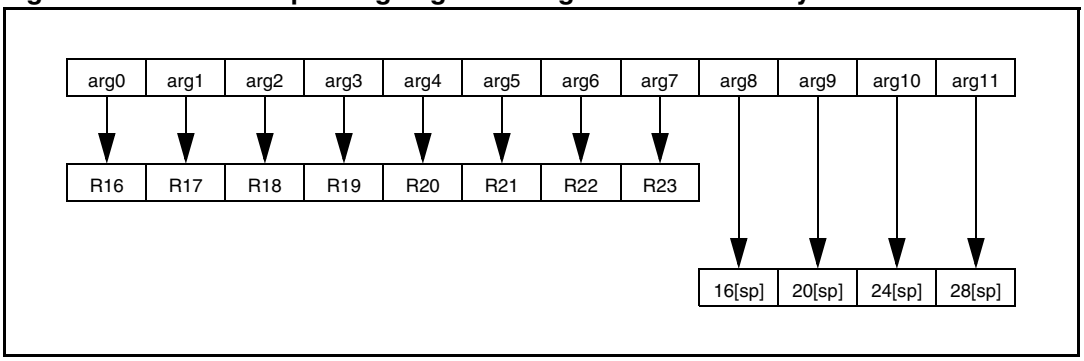
Parameters are passed in a combination of general registers and memory, as described below, and as illustrated in *Figure 12*.

The first 32 bytes of the parameter list are passed in registers, and the rest of the parameters are passed on the memory stack, beginning at the caller's stack pointer plus 16 bytes. The caller uses up to 8 general registers.

To accommodate the variable argument lists in the C language, there is a fixed correspondence between an argument's position in the parameter list and the register used for general register arguments. This allows a procedure to spill its register arguments easily to memory before stepping through the argument list with a pointer. Since the ST200 architecture does not have floating-point registers, there is no issue whether floating-point parameters should be passed in both general registers and floating-point registers.

The following sections explain these conventions in detail.

Figure 12. Parameter passing in general registers and memory



7.4.1 General rules

Parameters are first allocated on a conceptual argument list, which is then mapped to a combination of registers and the memory stack. The argument list begins at relative address 0, and is allocated towards increasing addresses. The argument list begins at a double word boundary. Each parameter begins on a 32-bit (4 byte) boundary. Each 32-bit storage unit in the argument list is called an argument slot. The argument slots are named `arg0`, `arg1`, `arg2`, …, as shown in *Figure 12*. Some parameters may use more than one argument slot.

7.4.2 Alignment and padding

Parameters are aligned and padded within the argument list according to the following rules.

- Integral scalar parameters smaller than 32 bits are padded on the left to a total width of 32 bits (that is, the value is placed in the least-significant bits of the argument slot), and placed in the next available argument slot. The contents of the padding bits are undefined.
- 32-bit integral scalar parameters and pointers are placed in the next available argument slot.
- 64-bit integral scalar parameters are aligned to an 8-byte boundary (possibly leaving an argument slot empty), and placed in the next two consecutive argument slots. For little endian, the low-order 32 bits of the parameter are passed in the first argument slot. For big endian parameters placed within the first 8 argument slots, the low-order 32 bits of the parameter are passed in the first argument slot. For big endian parameters placed after the first 8 argument slots, the high-order 32 bits of the parameter are passed in the first argument slot.
- Single-precision (32-bit) floating-point scalar parameters are placed in the next available argument slot.
- Double-precision (64-bit) floating-point scalar parameters are aligned to an 8-byte boundary (possibly leaving an argument slot empty), and placed in the next two consecutive argument slots. For little endian, the low-order 32 bits of the parameter are passed in the first argument slot. For big endian parameters placed within the first 8 argument slots, the low-order 32 bits of the parameter are passed in the first argument slot. For big endian parameters placed after the first 8 argument slots, the high-order 32 bits are passed in the first argument slot.
- Single-precision complex scalar parameters are passed as a pair of single-precision floating-point scalar parameters. The real part is passed first.
- Double-precision complex scalar parameters are passed as a pair of double-precision floating-point scalar parameters. The real part is passed first.
- Aggregates (structures and arrays, including complex numbers) up to 4 bytes in size, when passed by value, are padded to a total width of 32 bits and placed in the next available argument slot. If shorter than 4 bytes, the aggregate begins at the lowest-numbered byte in the storage unit, so for little endian environments, the padding is on the left, and for big endian environments, it is on the right.
- Aggregates larger than 4 bytes, when passed by value, are padded to a multiple of 32 bits and aligned to an 8-byte boundary (possibly leaving an argument slot empty), and placed in as many argument slots as needed. For little endian environments, any padding required is on the left in the final argument slot, and for big endian environments, it is on the right in the final argument slot.

7.4.3 Register parameters

The first eight argument slots (32 bytes) in the argument list are passed in registers, according to the following rules.

- The eight argument slots are associated, one-to-one, with the procedure argument general registers, as shown in *Figure 12 on page 30*.
- If an aggregate parameter straddles the boundary between `arg7` and `arg8`, the part that lies within the first eight slots is passed in general registers, and the remainder is passed in memory, as described in *Section 7.4.4*.

7.4.4 Memory stack parameters

The remainder of the parameter list, beginning with `arg8`, is passed in the outgoing parameter area of the memory stack frame, as described in *Section 6.1: Procedure frames on page 24*. Parameters are mapped directly to memory, with `arg8` placed at location SP+16, `arg9` at SP+20, and so on. Each argument slot is stored in memory as a 32-bit storage unit according to the byte order of the current environment.

7.4.5 Variable argument lists

The rules above support variable-argument list functions in both the K&R and the ANSI-C languages. Since the ST200 architecture does not have floating-point registers, there is no issue whether floating-point parameters should be passed in both general registers and floating-point registers, and whether the presence or absence of a function prototype changes the way floating-point arguments are passed.

Thus, a function with variable arguments may assume that the variable arguments that lie within the first eight argument slots can all be found in the argument passing general registers, R16-R23. It may then store these registers to memory, using the 16-byte scratch area for R20-R23, and using up to 16 bytes at the base of its own stack frame for R16-R19, as necessary. This arrangement places all the variable parameters in one contiguous block of memory.

In a big endian environment, the alignment and padding rules require the code that steps through the argument list to distinguish between aggregates and integers smaller than 4 bytes. Aggregates will be left-aligned within a 4-byte slot, while integers will be right-aligned.

In a big endian environment only, the ordering of 64-bit scalars and double-precision complex scalars requires that the code distinguishes between double precision values passed in registers (in the first eight argument slots) and passed in memory. For a 64-bit value passed in a pair of registers, the low-order 32 bits are always passed in the lower numbered register, which will appear at the most negative memory address when the variable arguments are stored to memory. For a double precision value passed in memory, the high-order 32 bits will appear at the most negative memory address.

7.4.6 Languages other than C

Most languages other than C can usually be treated as if prototypes are always in scope. However, a compiler for another language may need to honor the variable argument list conventions, if it provides a mechanism for calling C procedures that may have variable argument lists.

7.4.7 Examples

The following examples illustrate the parameter passing conventions.

Figure 13. Scalar integers and floats, with prototype

```
extern int func(int,double,double,int);

func(i,a,b,j);
```

The parameters are passed as follows: `i` in R16; `a` in R18 and R19; `b` in R20 and R21; and `j` in R22.

Figure 14. Scalar integers and floats, with prototype

```
extern int func(int,double,double,int);

func(i,a,b,j);
```

The parameters are passed as follows: `i` in R16; `a` in R18 and R19; `b` in R20 and R21; and `j` in R22.

Figure 15. Scalar integers and floats, without prototype

```
extern int func();

func(i,a,b,j);
```

The current ST200 architecture does not define separate floating-point hardware. Hence the parameters are passed as shown in *Figure 14*.

Figure 16. Aggregates

```
struct s { char c; int i; double d; };

int func(int i, struct s a);
```

The parameters are passed as follows: `i` in R16, `a.c` in R18, `a.i` in R19, and `a.d` in R20 and R21.

7.5 Return values

Values up to 32 bytes and certain aggregates are returned directly in registers, according to the following rules.

- Integral and floating-point values up to 32 bits in size are returned in R16. For integers smaller than 32 bits, the contents of the upper bits must be zero-filled (if unsigned) or sign-extended (if signed) to 32 bits.
- 64-bit integral and floating-point values are returned in R16 and R17.The low-order 32 bits of the result are returned in R16.
- Single-precision complex values are returned as a pair of single-precision floating-point values, the real part in R16, the imaginary part in R17.
- Double-precision complex values are returned as a pair of double-precision floating-point values: the low-order 32 bits of the real part in R16, the high-order 32-bits of the real part in R17, the low-order 32 bits of the imaginary part in R18, the high-order 32 bits of the imaginary part in R19.

Aggregates are padded to a multiple of 32 bits, and returned in successive general registers, beginning with R16. Values up to 32 bits use R16, values from 32 to 64 bits use R16 and R17, and so on. For little endian environments, padding is to the left in the last (highest-numbered) register; for big endian environments, padding is to the right in the last (highest-numbered) register. The padding is neither required nor guaranteed to be zeroes.

Return values larger than 32 bytes are returned in a buffer allocated by the caller. A pointer to the buffer is passed to the called procedure in R15. This register is not guaranteed to be preserved by the called procedure (that is, the caller must preserve the address of the buffer through some means).

A procedure may assume that any procedure it calls will return a valid value.

7.6 Requirements for unwinding the stack

Certain constraints must be met in order to unwind the stack successfully at any time, both by standard procedure calls as described here, and by special-purpose calling conventions. To meet the needs of the stack unwind mechanism, the following rules must be followed at all times.

- The link register must be preserved prior to any call. The compiler must record where the link register is saved and over what range of code the saved value is valid.
- If the procedure has a memory stack frame, the compiler must record either how large the frame is; or that a previous frame pointer is stored on the stack or in a general register.

7.7 Rationale

This section provides the rationale for various linkage conventions.

7.7.1 Parameter passing conventions

The ST200 architecture does not have floating-point registers. Hence there is no issue whether floating-point parameters should be passed in both general registers and floating-point registers. However, the register parameter convention could be easily extended to cover floating-point parameters if the ST200 architecture were extended to have floating-point registers.

7.7.2 Passing structures by value

Structures are not decomposed in general, since that would complicate variable argument lists and taking the address of a formal parameter.

7.7.3 Variable argument lists

The **varargs** convention is designed to make variable arguments list relatively efficient without imposing too much of a burden on code that does not use **varargs**. An alternative convention is to assign arguments to general registers according to the type of the actual parameter, without leaving holes. The **varargs** procedure is then responsible for saving the eight general registers in a save area. It would need to maintain two pointers – one to the register area, plus one for the parameters passed in memory – and step through these areas based on the types supplied to the `va_arg` macro. The proposed convention is more efficient than this alternative for **varargs** handling.

7.7.4 Shrink-wrapping

Because there are so few registers that must be saved explicitly at procedure entry, shrink-wrapping register saves are probably not necessary. However, it may prove desirable to be able to shrink-wrap the frame creation, so that a procedure with a quick exit can avoid creating the frame on that path.

7.7.5 Tail calls

A compiler can choose to make a tail call (including tail recursion) by simply branching (without linking) to the procedure being called. However, this appears to a debugger as if the nested procedure was called directly from the caller of the original procedure.

If the calling procedure has a memory stack frame, it must pop its frame off the stack before making the call. Parameters should be passed in the usual registers.

Tail calls can only be made when all parameters can be passed in registers.

7.7.6 Intrinsic functions, built-ins, special calling conventions

"**Millicode**" procedures and other types of compiler-supported routines may use special conventions known to the compiler, and destroy only a subset of the scratch registers. Thus a procedure may keep values in certain scratch registers across millicode calls, and can still be considered a leaf procedure if it makes no other calls. Each millicode procedure specifies its parameter passing conventions and the set of scratch registers that are destroyed. In addition, some millicode procedures may have known execution times so that they may be used effectively in a pipelined loop. Because this type of procedure call is a special convention, involving only the compiler and run-time library, it is outside the scope of this document. The run-time architecture requires only that any such special-purpose calls satisfy the requirements for unwinding the stack, as discussed in *Section 7.6 on page 34*.

While it may be desirable to establish a standard library of millicode routines that can be shared by all compilers, such a specification should be a supplement to this document, rather than part of it.

7.7.7 Use of the GP register

There are many ways to architect the use of the GP register. To provide a single ABI suitable both for standalone embedded applications and for sophisticated OS environments supporting shared libraries, it is necessary that the representation of a function pointer does not change in PIC code, that is, a function pointer is still a pointer to the code of the function. This prohibits techniques that use function descriptors and have the caller set up the GP for the callee. The chosen technique requires a procedure to establish a global data pointer for its own use, using PC-relative addressing to find the data segment. This means that the offset between text segment and data segment must be a constant value established at link time.

The GP register could be either preserved or scratch. If it is a scratch register, there is no need to save and restore it in the function prologue and epilogue, but it needs to be re-materialized following a procedure call. Choosing a preserved register means that it does not need to be re-materialized following procedure calls. However, this means that any function that establishes a GP value must save and restore the GP register. The sequence to establish a GP value also overwrites the link register, so this register must also be saved and restored. A leaf function is not required to use the GP register to hold the GP value, it can use a scratch register instead, to avoid having to save and restore the GP register.

Internal functions (those functions that can only be called from within the same load module) can assume that the GP register is already correctly set, and omit the code to set the GP register. Historically, separate compilation and the global visibility rules of C have been obstacles to the detection of internal functions, but more recent functionality such as ELF visibility attributes and linker version scripts provide the compiler with the means to detect them. It is anticipated that compiler interprocedural analysis will be applied to a whole load module, to automatically discover the internal functions (a discovered internal function is a function whose entry-point is not exported and whose address is never taken).

7.7.8 Function pointers

An alternative design that was considered is to define function pointers as pointers to either the actual function entry point or to an *export stub*. For functions that do not need a GP value, or that belong to a statically bound program, the function pointer would always point directly to the entry point. For functions that need a GP value, the function pointer would point to an export stub that would materialize the GP value before branching to the actual entry point; the export stub would be allocated in the target procedure's data segment, so that the DLL loader can initialize the proper GP value in the stub itself. This makes the point of call as short as possible, and optimizes not only all indirect calls in statically bound programs but also indirect calls to many leaf procedures in dynamically bound programs (those that do not reference any static data).

In this alternative design, we felt that the extra branching through the export stubs and the TLB pollution caused by both data and instruction references to the same area would cost too much.

8 Coding conventions

This chapter discusses general coding conventions and presents some example code sequences for various tasks.

The code sequences shown in this chapter are intended to serve as guidelines and examples rather than as required coding conventions. The requirements are documented in other chapters of this document.

8.1 Sample code sequences

In the sample code sequences in this section, registers of the form T1, T2, and so on, are temporary registers, and may be assigned to any available scratch register. The code sequences show necessary cycle breaks but not delay slots, and no other scheduling considerations have been made. It is assumed that these code sequences will be scheduled with surrounding code to make best use of the processor resources.

The following assembly language operators are used:

@gprel(*expr*) Computes a gp-relative displacement: the difference between *expr* and the value of the global pointer (GP) for the current module.

@gotoff(*expr*) Computes the gp-relative displacement to the global offset table entry for *expr*.

@neggprel(*expr*) Computes the difference between the value of the global pointer for the current module and *expr*, that is, the negative of **@gprel(*expr*)**.

8.1.1 Position-independent function prologue

A position-independent function that needs the GP value may calculate this value on entry, in the function prologue. Normally the GP value is placed in register R14, which is a preserved register, so it does not need to be re-materialized after function calls. In a leaf function, the GP value may be placed in any general register.

Table 5 illustrates the code used to calculate the GP value into register R14.

Table 5. Calculation of GP value

When **addpc** operation is not available	`call $r63=.lab`
	`;;`
	`.lab:add $r14=$r63,@neggprel(.lab)`

When **addpc** operation is available	`.lab:addpc $r14=@neggprel(.lab)`

If the **addpc** operation is not available on the target, then a **call** operation must be used. The **call** operation puts the address of the next bundle into R63. The **add** operation then calculates the GP value into R14: R63 contains the value .lab, `@neggprel(.lab)` gives the value `GP - .lab`, so the **add** operation calculates `.lab + (GP - .lab)`, yielding the required GP value.

If the **addpc** operation is available on the target it should be used in preference, because this usage of the **call** operation can make branch prediction less effective.

This sequence can be added to the standard function prologue, giving a standard function prologue for position-independent code.

Figure 17 illustrates a function prologue that allocates 64 bytes of local stack space and saves registers R8, R9 and R63.

Note: R14 *must also be saved.*

Figure 17. Position-independent function prologue

```
prologue:
        stw 12[$r12]=$r63
        add $r12=$r12,-64
        ;;
        stw 72[$r12]=$r14
        call $r63=.L1
        ;;
.L1:    stw 68[$r12]=$r8
        add $r14=$r63,@neggprel(.L1)
        ;;
        stw 64[$r12]=$r9
```

Figure 18 illustrates how the prologue can be optimized for a leaf function that requires the GP value, but does not need to allocate any local stack space or save any registers.

Note: The GP value is established in scratch register R9, to avoid the need to save and restore R14.

Figure 18. Position-independent leaf function prologue

```
prologue:
        mov $r8=$r63
        call $r63=.L1
        ;;
.L1:    add $r9=$r63,@neggprel(.L1)
        mov $r63=$r8
```

8.1.2 Addressing data in the data area

In absolute code, data may be addressed with an absolute address, as illustrated in *Figure 19*.

Figure 19. Absolute data access

```
ldw $r24=var # load contents of var
```

In position-independent code, “own” data may be addressed with a simple direct reference relative to the GP register, as illustrated in *Figure 20*.

Figure 20. GP-relative data access

```
ldw $r24=@gprel(var)[$r14] # load contents of var
```

In position-independent code, data that is not known to be defined in the current load module (it is not “own”) must be accessed indirectly through the global offset table, as illustrated in *Figure 21*.

Figure 21. GOT-based data access

```
ldw t1=@gotoff(var)[$r14] # load address of var from GOT entry
;;
...
ldw $r24=[t1] # load contents of var
```

8.1.3 Addressing literals in the text segment

In absolute code, literals in the text segment may be addressed with an absolute address, as shown in *Figure 19*.

In position-independent code, literals in the text segment may be addressed with GP-relative addressing, as shown in *Figure 20*.

8.1.4 Materializing function pointers

Function pointers may be obtained from the data segment as an initialized word (*Figure 22*). This word may be accessed as described in *Section 8.1.2*.

Figure 22. Initialized function pointer in data segment

```
fptr: .data4 func # initialize function pointer
```

In absolute code, function pointers may be obtained through absolute addressing.

Figure 23. Absolute function pointer materialization

```
mov $r24=func # get address of function in R24
```

In position-independent code, a pointer to an “own” function may be obtained through GP-relative addressing (*Figure 24*). A pointer to a function that is not known to be defined in the current load module must be obtained using the global offset table (*Figure 25*).

Figure 24. GP-relative function pointer materialization

```
add $r24=$r14, @gprel(func) # get address of function in R24
```

Figure 25. GOT-based function pointer materialization

```
ldw $r24=@gotoff(func)[$r14] # get address of func in R24
```

8.1.5 Direct procedure calls

The following code sequence assumes that the parameters have already been placed in the proper locations.

Figure 26. Direct procedure call

```
call $r63=func # make the call
```

The same code sequence can be used even if the function being called is in a different load module. In this case, the linker will create a stub to redirect the call through an address in the global offset table. The linker relocates the original call instruction to call this stub. Note the stub assumes that register R14 contains the GP value. *Figure 27* gives an example of the stub.

Figure 27. Linkage stub for direct call to another load module

```
.plt.func:
        ldw $r63=.got.plt.func[$r14] # load address of func
        mov $r9=$r63 # save R63 temporarily
        ...
        goto $r63 # jump to func
        mov $r63=$r9 # restore R63
```

A direct call operation has a range of 16MBytes. For calls to more distant functions in the same load module, the linker can introduce a long branch stub. The stub is placed within 16MBytes of the call. The linker resolves the call so that it calls the stub, and the stub transfers control to the called function.

Table 6 gives an example of the stub contents.

Table 6. Linkage stub for long direct call to `func`

Absolute		`mov $r63=func`
		`mov $r9=$r63`
		`;;`
		`...`
		`goto $r63`
		`mov $r63=$r9`

Position-independent, when **addpc** operation is not available	`.lab:`	`call $r63=.lab`
		`mov $r9=$r63`
		`;;`
		`add $r63=$r63,func-.lab`
		`;;`
		`...`
		goto $r63
		mov $r63=$r9

Position-independent, when **addpc** operation is available	`.lab:`	`addpc $r63=func-.lab`
		`mov $r9=$r63`
		`;;`
		`goto $r63`
		`mov $r63=$r9`

8.1.6 Indirect procedure calls

The indirect procedure call sequence is similar to the direct procedure call sequence, above. In this example, the function pointer is assumed to have been loaded into the link register LR.

Figure 28. Example 6

```
call $r63=$r63 # make the call
```

8.1.7 Jump tables

High-level language constructs such as case and switch statements, where there are several possible local targets of a branch, may use a number of different code generation strategies, ranging from sequential conditional branches to a direct-lookup branch table.

In absolute code, if the compiler chooses to generate a branch table, the table should be placed in the data segment, and each table entry should be the absolute address of the branch target for that entry.

A sample indirect branch is shown in *Figure 29*. The branch table is assumed to be an array of 32-bit entries, each of which is the absolute address of a branch target. The branch table index is assumed to have been computed or loaded into register R24.

Figure 29. Absolute branch table

```
sh2add t1=$r24,brtab  # calculate address of branch table entry
 ;;
 ...
 ldw $r63=[t1] # load branch table entry
 ;;
 ...
 goto $r63 # … and branch

 ...
 .data
 ...
.brtab:
  .word case0
 .word case1
 .word case2
 ...
```

In position-independent code, the same basic idea will work, provided that GP-relative addressing is used to calculate the address of the branch table (*Figure 30*).

Figure 30. Position-independent branch table

```
 add t1=$r14,@gprel(brtab)
 ;;
 sh2add t1=$r24,t1  # calculate address of branch table entry
 ;;
 ...
 ldw $r63=[t1] # load branch table entry
 ;;
 ...
 goto $r63 # … and branch
 ...
 .data
 ...
.brtab:
  .word case0
 .word case1
 .word case2
 ...
```

However, this will require a dynamic relocation for each entry in the branch table. This can be avoided by making the branch table a table of branch instructions and placing it in the text segment. This also means that the branch table can be shared between processes (*Figure 31*).

Figure 31. Improved position-independent branch table

```
 add t1=$r14,@gprel(brtab)
 ;;
 sh2add $r63=$r24,t1  # calculate address of branch table entry
 ;;
 ...
 goto $r63 # … and branch
 ...
.brtab:
 goto case0
   ;;
 goto case1
   ;;
 goto case2
   ;;
```

8.2 Up-level referencing

Local variables visible to nested procedures must be saved in memory at any procedure call or exception control point; a procedure's local registers are not visible to its nested procedures.

These conventions suggest, but do not require, the use of a static link passed as an implicit parameter to nested procedures. The static link can be used by the nested procedure to access local variables in its enclosing scope. The rules for forming and passing static links are as follows.

- A level-one procedure (outermost) calling a level-two procedure should pass, as the static link, the address of a known reference point within its stack frame (for example, its frame pointer).
- A nested procedure calling another procedure at the same level should pass, as the static link, the static link that it received.
- A nested procedure calling a procedure nested within it should store the static link that it received at a known place within its own stack frame, then pass, as the static link to the new procedure, the address of a known reference point within its own stack frame (for example, a pointer to the static link that it saved).
- A nested procedure calling a less-deeply nested procedure must follow the chain of static links to obtain the correct static link to pass.

When forming function pointers that refer to nested procedures, the same rules apply. The static link must be determined at the time the function pointer is materialized, and stored with the function pointer.

To reference local variables in enclosing scopes, the chain of static links must be followed to obtain a pointer to the enclosing scope's stack frame. The compiler can determine statically the offset of the desired local variable relative to the reference point used for the static link.

An alternative implementation is a display pointer, also passed as an implicit parameter to each nested procedure.

9 Context management

This chapter presents some rules for context management.

9.1 Process and thread context

The following resources constitute the context that is visible to the user-mode process or thread (not including the program's address space). These are the registers that must be saved and restored on any form of context switch.

- Program counter (PC).
- General registers (R1-R63).
- Branch registers (B0-B7).

9.2 User-level thread switch, co-routines

Thread switches and co-routine calls can be done with a procedure call, so no scratch registers need to be saved as part of the context. The first part of this routine saves the current thread context on the stack.

1. Save all registers that must be preserved explicitly on the current stack.
2. Save the link register on the stack.

At this point, the memory stack pointer (SP) can be changed to point to the new thread's stack, and restore the new thread's context from there.

1. Restore the link register from the new stack.
2. Restore the explicitly preserved registers.
3. Switch to the new thread.

9.3 setjmp and longjmp

The `jmpbuf` structure used by `setjmp` and `longjmp` and declared in `<jmpbuf.h>`, needs to contain the following saved state:

- program counter (PC) – the link register from the call to `setjmp`(),
- memory stack pointer (SP),
- the preserved general registers, R1 - R7 and R14,
- the thread pointer (TP) register, R13.

Saving and restoring this context is similar to a thread switch.

10 Dynamic linking

This chapter presents an overview of the support for dynamic linking.

10.1 Position-independent code

All code within a dynamic link library (DLL) should be position independent (PIC). This allows the text segment of the DLL to remain pure so that it can be shared among many processes. Position-independence imposes two requirements on generated code:

- Code that forms an absolute address referring to any address in the DLL's text or data segments is not allowed, because the code would have to be relocated at load time, making it non-sharable. All branches must be PC-relative, references to the data segment and to constants and literals in the text segment must be relative to a base pointer (typically GP).
- Code that references symbols that are or may be imported from other load modules must use indirect addressing through a global offset table. The linker is expected to resolve procedure calls by creating import stubs, but the compilers must generate indirect loads and stores for data items that may be dynamically bound. In both cases, the indirection is made through the global offset table, allocated by the linker, and initialized by the dynamic loader. The global offset table is described in *Section 10.1.1* through to *Section 10.1.4*.

10.1.1 Procedure calls and long branch stubs

Normal procedure calls can be prepared with the call instructions, which use PC-relative addressing. There are three possible cases at link time:

- If the target is not within the same load module, or if it is subject to pre-emption by an earlier definition from another load module, the linker must allocate an import stub and resolve the call instruction to the stub.
- If the target is known to be within the same load module and the displacement is small enough, the call instruction can be statically resolved to the call target.
- If the target is within the same load module, but the displacement is too far for the call instruction, the linker must allocate a long branch stub. The long branch stub itself must satisfy the PIC requirements. If the target is within range of the stub, the stub may use a PC-relative **goto** instruction; otherwise, it must load the address of the target from the global offset table.

10.1.2 Access to the data segment

The DLL's data segment must be accessed through the GP value, which must be established by a DLL procedure before use. The GP value is used to access both the global offset tables and statically allocated data.

There are several cases here:

- Global variables that are imported from another load module, or that are subject to pre-emption by an earlier definition in another load module, must be accessed indirectly through the global offset table. The compiler must generate code to load a pointer from the global offset table, using GP-relative addressing, and then access the data item using that pointer. The compiler does not have to allocate the global offset table; there are relocations defined in the object file format that instruct the linker to allocate a global offset table slot and to supply the GP-relative address of that slot.
- Statically allocated variables of local scope, or global variables whose definitions are not subject to pre-emption, may be accessed directly with GP-relative addressing.

10.1.3 Access to constants and literals in the text segment

Constants and literals allocated in the text segment may be accessed with GP-relative addressing, or with indirect addressing through the global offset table.

10.1.4 Materializing function pointers

Function pointers may be materialized by indirect addressing through the global offset table. Pointers to functions that are not subject to preemption may be materialized using GP-relative addressing.

Function pointers may not be materialized from immediate operands.

10.2 Import stubs

When the linker determines that a procedure call refers to an entry point in a different load module, it resolves the reference locally by building an import stub with the same name as the intended target. The import stub contains code that obtains the entry point from the global offset table, then transfers control, as described in *Section 7.3: Calling sequence on page 27*.

If the compiler is provided with enough information to know that a particular entry point is in a different load module, it may generate a calling sequence that obviates the need for the linker to build an import stub. However, this calling sequence is ABI specific, and is not specified in this document.

10.3 The dynamic loader

The dynamic loader is a component of the operating system software that locates all load modules belonging to an application, loads them into memory, and binds the symbolic references among them. Most of the operation of the dynamic loader is specific to the particular operating system environment, and is further described in the ABIs for those environments. The common run-time architecture has been designed to minimize the amount of work involved in the binding process, by concentrating most of the relocation required in the global offset tables, and by prohibiting any items in the text segment that may require dynamic relocation.

10.4 Rationale

Code in main programs may be absolute or position independent. If an absolute program imports data from a DLL, the linker is forced to allocate the data in the main program's data segment statically (this is commonly called the "`.dynbss` hack"). When data imported from DLLs is allocated in the main program's data segment, the program may be subject to future compatibility problems when the DLL is replaced with a newer version. This issue may be avoided by requiring main programs to be position independent, at the cost of some efficiency in the main program. This compatibility/performance trade-off is not made in the common run-time architecture, it is left to the specific ABI.

11 System interfaces

This chapter presents an overview of what an application expects during its system-dependent execution.

11.1 Process initialization

An application begins its execution at a specified program entry point, which depends on the primary language in which the application is written. For C programs, the function main is the program entry point. However, on most operating systems some system-dependent initialization must take place before control is transferred to this entry point. This initialization may take place in the operating system or in the dynamic loader.

This section presents a general overview of what an application expects when its program entry point receives control. The ABI document for each operating system is expected to contain the details.

11.1.1 Initial memory stack

The memory stack pointer (SP), must be properly aligned, and must contain an address that is suitable for allocation of the program's first stack frame. There must be a 16-byte scratch area available for use, beginning at the address in SP, but the application may make no further assumptions about the contents of the stack beyond the scratch area.

11.1.2 Initial register values

The stack pointer register must be initialized correctly, as described above.

The content of the parameter registers, R16–R23, are system-dependent, and are typically used for transmitting the program arguments.

There is no requirement to initialize the global pointer register R14: the GP value should be materialized by those functions that require it.

11.2 System calls

System API routines are called using the standard calling conventions described in *Chapter 7: Procedure linkage on page 27*. Any special interfaces between these API routines and the operating system itself are system-dependent.

11.3 Traps and signals

When the operating system delivers a signal or an exception to a user process, it must make the following available to the process:

- a context record, containing the full user-visible context
- the cause of the trap. If the trap was caused by an instruction, the information must be sufficient to identify the instruction bundle

When a trap or signal handler returns, OS help is necessary for restoring the complete context (using **rfi**). Thus, the OS must build a dummy stack frame for the handler, so that a return from the handler will transfer to an OS entry point that can restore the full context.

Appendix A Standard header files

The standard header files (`*.h`) delivered with the ST200 tools are located in the `<tools-dir>/include` directory.

A.1 Implementation limits

The following constants are defined in the `<limits.h>` header file:

```
#define CHAR_BIT                8
#define SCHAR_MIN               (-128)
#define SCHAR_MAX               127
#define UCHAR_MAX               255
/* MB_LEN_MAX determined by locale information */
#define CHAR_MIN                SCHAR_MIN
#define CHAR_MAX                SCHAR_MAX
#define SHRT_MIN                (-32768)
#define SHRT_MAX                32767
#define USHRT_MAX               65535
#define INT_MIN                 (-2147483647-1)
#define INT_MAX                 2147483647
#define UINT_MAX                4294967295U
#define LONG_MIN                (-2147483647L-1)
#define LONG_MAX                2147483647L
#define ULONG_MAX               4294967295UL
#define LLONG_MIN               -9223372036854775808LL
#define LLONG_MAX               9223372036854775807LL
#define ULLONG_MAX              18446744073709551615ULL
```

A.2 Floating-point definitions

The following constants are defined in the <float.h> header file:

```
#define FLT_RADIX                2
#define FLT_ROUNDS               1
#define FLT_EPSILON              1.19209290E-07F
#define FLT_DIG                  6
#define FLT_MANT_DIG             24
#define FLT_MIN                  1.17549435E-38F
#define FLT_MIN_EXP              (-125)
#define FLT_MIN_10_EXP           (-37)
#define FLT_MAX                  3.40282347E+38F
#define FLT_MAX_EXP              128
#define FLT_MAX_10_EXP           38
#define DBL_EPSILON              2.2204460492503131E-16
#define DBL_DIG                  15
#define DBL_MANT_DIG             53
#define DBL_MIN                  2.2250738585072014E-308
#define DBL_MIN_EXP              (-1021)
#define DBL_MIN_10_EXP           (-307)
#define DBL_MAX                  1.7976931348623157E+308
#define DBL_MAX_EXP              1024
#define DBL_MAX_10_EXP           308
```

A.3 Variable argument list macros

This is an example implementation of the variable argument list macros provided in the <stdarg.h> header file. Similar definitions for K&R may be found in <varargs.h>.

```
#define __va_align(list,size) \
    (((int) (list) + (size) - 1) & ~((size) - 1))

#define __va_rounded_size(type) \
     __va_align(sizeof(type),sizeof(int))

#define va_end(list)(void)0

#ifdef __LITTLE_ENDIAN__

typedef char *va_list;

#define va_start(list,parmN)(list = (va_list)__va_align(&parmN + 1,4))
```

```
#define va_arg(list, type)\
   (list = (va_list) (__va_align((((type *) \
                              __va_align((list), \
                                    sizeof(type) > 4 ? 8 : 4)) + 1), 4)), \
   *((type *) ((int) (list) - __va_rounded_size (type))))

#define va_copy(dest,src) ((dest) = (src))

#else
/* BIG ENDIAN */

typedef struct {
   char *__next;
   char *__reg_limit;
} va_list[1];

#define va_start(list,parmN)  \
   ((list)->__next = ((char *)__va_align(&(parmN) + 1, 4), \
    (list)->__reg_limit = __va_reg_limit()))

#define __va_fetch_64bit_scalar(p,lim,type) \
        ((((p) - 8) < (lim)) \
          ? ({ union { type __d; \
                       struct { unsigned int __hi, __lo; } __s; } __u; \
               __u.__s.__hi = *(int *)((p) - 4); \
               __u.__s.__lo = *(int *)((p) - 8); \
               __u.__d; }) \
          : *((type *)((p) - 8)))

#define __va_fetch_64bit_complex(p,lim,type) \
          ({ union { type __d; \
                     struct { double __r; double __i; } __s; } __u; \
             __u.__s.__r = __va_fetch_64bit_scalar((p)-8,(lim),double);\
             __u.__s.__i = __va_fetch_64bit_scalar((p),(lim),double); \
             __u.__d; })

#define va_arg(list, type)          \
   ((list)->__next = (void *)(__va_align((((type *) \
      __va_align((list)->__next, (sizeof(type) > 4) ? 8 : 4)) + 1), 4)), \
         (__va_64bit_scalar(type) \
           ? __va_fetch_64bit_scalar(((int)(list)->__next), \
                                     ((int)(list)->__reg_limit), type):\
          __va_64bit_complex(type) \
           ? __va_fetch_64bit_complex(((int)(list)->__next),\
                                     ((int)(list)->__reg_limit), type):\
          (*((type *) ((int) (list)->__next - \
                 ((sizeof (type) < 4 && !__va_aggregate(type))\
                          ? sizeof (type) \
                          : __va_rounded_size (type)))))))

#define va_copy(dest, src) memcpy(dest, src, sizeof(va_list))
#endif
```

The big endian versions of the `va_start` and `va_arg` macros are more complicated because when the argument registers are saved to memory, 64-bit scalars are not stored in their natural ordering. `va_arg` requires a number of compiler builtin-functions.

- `__va_aggregate(m)` returns true if `m` is an aggregate type.
- `__va_64bit_scalar(m)` returns true if `m` is a 64-bit scalar type (double-precision floating-point or long long integer).
- `__va_64bit_complex(m)` returns true if `m` is a double-precision complex type.
- `__va_reg_limit()` returns the address at which the in-memory representation of 64-bit scalars passed as variadic arguments changes. This is the address of the first byte above the argument register save area. (In the case that there is no argument save area, for example, when all the argument registers contain fixed arguments, then `__va_reg_limit` may return any value within the callee's stack frame, since all variadic arguments will be in the caller's stack frame.)

A language extension to allow declarations to be nested within expressions has been used in the `__va_fetch_64bit_scalar` and `__va_fetch_64bit_complex` macros. This could be avoided by adding the variable declarations to the `va_list` type.

A.4 setjmp and longjmp

The following definition is provided in the `<setjmp.h>` header file:

```
typedef long long jmp_buf[_JBLEN];
```

The jump buffer is defined to be long enough to contain the context defined in *Section 9.3: setjmp and longjmp on page 44*.

The jump buffer must be declared to guarantee 8-byte alignment. The contents includes the following registers:

- program counter (PC) - the link register from the call to `setjmp()`,
- memory stack pointer R12 (SP),
- general registers R1-R7 and R14,
- reserved thread pointer register R13.

The size of the jump buffer (the value of `_JBLEN`) and the locations of individual items within the jump buffer are ABI specific.

Revision history

Table 7. Document revision history

Date	Revision	Changes
May 03	A	Initial release.
May 03	B	Data representation chapter: Scalar types table: Updated details for long long. Bit field ranges table: Updated entire table and removed following sentence.
Oct 03	C	Throughout: Link register (LR) is used in place of Return pointer (RP). Introduction chapter: Added new entries to Glossary. Memory model chapter: Updated introduction and Data allocation sections. Data representation chapter: Updated Bit-fields information. Register usage chapter: Registers $R0.13-$R0.14 are now reserved. System interfaces chapter: Updated details of Traps and signals. Standard header files appendix: Added details of limits.h, float.h, stdarg.h, varargs.h and setjmp.h files.
Jun 04	D	Throughout: Added references to ST231 and the ST231 Core and Instruction Set Architecture manual.
July 06	E	Added position independent code and dynamic linking. Modify 64-bit parameter passing for big-endian. Define $R0.13 as thread pointer. Define some sbrk operand values. Add usage of addpc to coding conventions. Add long branch stubs to coding conventions.
08-Nov-2006	F	Throughout: Corrected register naming conventions and assembler syntax.
11-Sep-2007	G	Reflects the R6.0 Toolset release. Updated *Preface on page 5* to list current document suite. Replaced "*ST200 Cross Development Manual 7521642"* with "*ST200 User Manual 8063762"* . Replaced "*ST200 Toolset User Manual 7508723*" with "*ST200 Compiler Manual 7508723*". Updated *Section 1.3: Glossary on page 9* to add defintion of millicode.

Table 7. Document revision history (continued)

Date	Revision	Changes
05-Dec-2007	H	Updated *Preface on page 5* to list current document suite, added where to access license information; added terminology section. Minor rewording of : *Chapter 1: Introduction on page 8*. *Section 3.2: Data allocation on page 16*. *Section 7.7: Rationale on page 34*. *Chapter 9: Context management on page 44*.
15-Jun-2009	I	Removed reference to ST220.
24-Oct-2012	10	Removed references to *ST240 core and instruction set architecture reference manual* as the ST240 core is no longer supported: – updated *ST200 documentation suite on page 5* – updated *Chapter 2: ST200 architecture on page 12*

Index

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W