M•CORE

Applications Binary Interface Standards Manual

TABLE OF CONTENTS

Paragraph **Page**

TABLE OF CONTENTS (Continued)

Paragraph **Page**

SECTION 5
LIBRARIES

TABLE OF CONTENTS (Continued)

Paragraph **Page**

TABLE OF CONTENTS (Continued)

Paragraph **Page**

LIST OF ILLUSTRATIONS

Number **Page**

LIST OF ILLUSTRATIONS (Continued)

Number **Page**

LIST OF TABLES

LIST OF TABLES (Continued)

Number **Page**

SECTION 1 INTRODUCTION

1.1 Scope

This manual defines the Motorola M•CORE Applications Binary Interface (ABI). The ABI is a set of interface standards that writers of compilers and assemblers must use when creating tools for the M•CORE architecture. These standards cover run-time aspects as well as object formats to be used by compatible tool chains. A standard definition ensures that all M•CORE tools are compatible and can interoperate.

Although compiler support routines are provided, this manual does not describe how to write M•CORE development tools, does not define the services provided by an operating system, and does not define a set of libraries. Those tasks must be performed by suppliers of tools, libraries, and operating systems.

1.2 Purpose

The standards defined in this manual ensure that all chains of development tools for the M•CORE will be fully compatible. Fully compatible tools can interoperate, and thus make it possible to select an optimum tool for each link in the chain, rather than selecting an entire chain on the basis of overall performance. The M•CORE Technology Center will provide a test suite to verify compliance with published standards.

The standards in this manual also ensure that compatible libraries of binary components can be created and maintained. Such libraries make it possible for developers to synthesize applications from binary components, and can make libraries of common services stored in on-chip ROM available to applications executing from off-chip ROM. With established standards, developers can build up libraries over time with the assurance of continued compatibility.

Two overriding goals are reflected in this manual:

- Use of interfaces that allow future optimizations for performance and energy.

  For example, whenever possible, registers are used to pass arguments, even though always using the stack might be easier. Small programs whose working sets fit into the registers are thus not forced to make unnecessary memory references to the stack just to satisfy the linkage convention.

- Use of interfaces that are compatible with legacy “C” code written for the M68000 whenever possible.

  For example, whenever possible, M68000 rules are used to build an argument list. This not only fits the M68000 programmer’s expectations, but easily supports old code that doesn’t use the ANSI standard macros for handling variable argument lists.

1.3 Overview

Standards in this manual are intended to preclude creation of incompatible development tools for the M•CORE, by ensuring binary compatibility between:

- Object modules generated by different tool chains
- Object modules and the M•CORE processor
- Object modules and source level debugging tools

Current definitions include the following types of standards.

1.3.1 Low-Level Run-Time Binary Interface Standards

- Processor specific binary interface — the instruction set, representation of fundamental data types, and exception handling
- Function calling conventions — how arguments are passed and results are returned, how registers are assigned, and how the calling stack is organized

1.3.2 Object File Binary Interface Standards

- Header convention
- Section layout
- Symbol table format
- Relocation information format
- Debugging information format

1.3.3 Source-Level Standards

- C language — preprocessor predefines, in-line assembly, and name mapping
- Assembler — syntax and directives

1.3.4 Library Standards

- Compiler assist libraries — floating point, and long-long integer

1.4 Associated Documentation

Please refer to the *M•CORE Reference Manual* (MCORERM/AD) for a detailed discussion of instruction set encoding and semantics.

1.5 Future Standards

This manual is meant to be expandable. Future standards may include the syntax used to insert assembly language statements into C language programs, and the syntax of a link editor command language.

SECTION 2 LOW-LEVEL BINARY INTERFACES

2.1 Underlying Processor Primitives

The complete M•CORE architecture is described in the *M•CORE Reference Manual* (MCORERM/AD).

2.1.1 Registers

The M•CORE ABI defines how to use the 16 general-purpose 32-bit registers of the M•CORE processor. These registers are named *r0* through *r15*.

The M•CORE can have up to 32 control registers. These registers are named *cr0* through *cr31*. The first 13 control registers are more commonly referred to by names that reflect their hard-wired function. The control registers are shown in **Table 2-1**.

The ABI does not mandate the semantics of the M•CORE Hardware Accelerator Interface (HAI) because these semantics vary between M•CORE implementations based on particular chips.

2.1.2 Fundamental Data Types

The M•CORE processor works with the following fundamental data types:

- unsigned *byte* of eight bits
- unsigned *halfword* of 16 bits
- unsigned *word* of 32 bits
- signed *byte* of eight bits
- signed *halfword* of 16 bits
- signed *word* of 32 bits

As the above list indicates, the data sizes are 8-bit *bytes*, 16-bit *halfwords* and 32-bit *words*. The mapping between these data types and the C language fundamental data types is shown in **Table 2-2**.

Table 2-1 M•CORE Control Registers

Register Use Convention		
Reg	**Name**	**Function**
cr0	psr, cr0	Processor Status Register
cr1	vbr, cr1	Vector Base Register
cr2	epsr, cr2	Shadow Exception PSR
cr3	fpsr, cr3	Shadow Fast Interrupt PSR
cr4	epc, cr4	Shadow Exception Program Counter
cr5	fpc, cr5	Shadow Fast Interrupt PC
cr6	ss0, cr6	Supervisor Scratch Register
cr7	ss1, cr7	Supervisor Scratch Register
cr8	ss2, cr8	Supervisor Scratch Register
cr9	ss3, cr9	Supervisor Scratch Register
cr10	ss4, cr10	Supervisor Scratch Register
cr11	gcr, cr11	Global Control Register
cr12	gsr, cr12	Global Status Register
cr13	cr13	Reserved
cr14	cr14	Reserved
cr15	cr15	Reserved
cr16	cr16	Reserved
cr17	cr17	Reserved
cr18	cr18	Reserved
cr19	cr19	Reserved
cr20	cr20	Reserved
cr21	cr21	Reserved
cr22	cr22	Reserved
cr23	cr23	Reserved
cr24	cr24	Reserved
cr25	cr25	Reserved
cr26	cr26	Reserved
cr27	cr27	Reserved
cr28	cr28	Reserved
cr29	cr29	Reserved
cr30	cr30	Reserved
cr31	cr31	Reserved

Table 2-2 Mapping of C Fundamental Data Types to the M•CORE

Fundamental Data Types			
ANSI C	**Size**	**Align**	**M•CORE**
char	1	1	unsigned byte
unsigned char	1	1	unsigned byte
signed char	1	1	signed byte
short	2	2	signed halfword
unsigned short	2	2	unsigned halfword
signed short	2	2	signed halfword
long	4	4	signed word
unsigned long	4	4	unsigned word
signed long	4	4	signed word
int	4	4	signed word
unsigned int	4	4	unsigned word
signed int	4	4	signed word
enum	4	4	signed word
data pointer	4	4	unsigned word
function ptr	4	4	unsigned word
long long	8	8	signed word:unsigned word
unsigned long long	8	8	unsigned word[2]
float	4	4	unsigned word
double	8	8	unsigned word[2]
long double	8	8	unsigned word[2]

Memory access to unsigned byte-sized data is directly supported through the ld.b (load byte) and st.b (store byte) instructions. Signed byte-sized access requires a sextb (sign extension) instruction after the ld.b. Access to unsigned halfword-sized data is directly supported through the ld.h (load halfword) and st.h (store halfword) instructions. Signed halfword access requires a sexth (sign extension) instruction after the ld.h. Memory access to word-sized data is supported through ld.w (load word) and st.w (store word) instructions. ld.w suffices for both signed and unsigned word access because the operation sets all 32 bits of the loaded register.

The M•CORE uses only big-endian byte ordering. The lowest addressable byte of a memory location always contains the most significant byte of the value. Fundamental data is always naturally aligned, i.e., a long is 4-byte aligned, a short is 2-byte aligned.

The M•CORE processor currently does not support the long long int data type with 64-bit operations. However, compliant compilers must emulate the data type. The long long int data type, both signed and unsigned, is eight bytes in length and 8-byte aligned.

Requiring long long int support as part of the ABI insures that the feature will exist in all tool chains, so that application developers can depend on its existence.

The M•CORE processor currently does not support floating point data. However, compliant compilers must support its use. The floating point format to be used is the IEEE standard for float and double data types. Support for the long double data type is optional but must conform to the IEEE standard format when provided.

Alignments are specifically chosen to avoid the possibility of access faults in the middle of an instruction (with the exception of load/store multiple).

2.1.3 Compound Data Types

Arrays, structures, unions, and bit fields have different alignment characteristics.

Arrays have the same alignment as their individual elements.

Unions and structures have the most restrictive alignment of their members. A structure containing a char, a short, and an int must have 4-byte alignment to match the alignment of the int field. In addition, the size of a union or structure must be an integral multiple of its alignment. Padding must be applied to the end of a union or structure to make its size a multiple of the alignment. Members must be aligned within a union or structure according to their type; padding must be introduced between members as necessary to meet this alignment requirement.

Bit fields cannot exceed 32 bits nor can they cross a word (32 bit) boundary. Bit fields of signed short and unsigned short type are further restricted to 16 bits in size and cannot cross 16-bit boundaries. Bit fields of signed char and unsigned char types are further restricted to eight bits in size and cannot cross 8-bit boundaries. Zero-width bit fields pad to the next 8, 16, or 32 bit boundary for char, short, and int types respectively. Outside of these restrictions, bit fields are packed together with no padding in between.

Bit fields are assigned in big-endian order, i.e., the first bit field occupies the most significant bits while subsequent fields occupy lesser bits. Unsigned bit fields range from 0 to $2^{w}–1$ where “w” is the size in bits. Signed bit fields range from $–2^{w–1}$ to $2^{w–1}–1$. Plain int bit fields are unsigned.

Bit fields impose alignment restrictions on their enclosing structure or union. The fundamental type of the bit field (e.g., char, short, int) imposes an alignment on the entire structure.

In the following example, the structure **`more`** has 4-byte alignment and will have size of four bytes because the fundamental type of the bit fields is int, which requires 4-byte alignment. The second structure, **`less`**, requires only 1-byte alignment because that is the requirement of the fundamental type (char) used in that structure. The alignments are driven by the underlying type, not the width of the fields. These alignments are to be considered along with any other structure members. **`Struct careful`** requires 4-byte alignment; its bit fields only require 1-byte alignment, but the field **`fluffy`** requires 4-byte alignment.

```
struct more {
     int first : 3;
     unsigned int second : 8;
};
struct less {
     unsigned char third : 3;
     unsigned char fourth : 8;
};
struct careful {
     unsigned char third : 3;
     unsigned char fourth : 8;
     int fluffy;
};
```

Fields within structures and unions begin on the next possible suitably aligned boundary for their data type. For non-bit fields, this is a suitable byte alignment. Bit fields begin at the next available bit offset with the following exception: the first bit field after a non-bit field member will be allocated on the next available byte boundary.

In the following example, the offset of the field “c” is one byte. The structure itself has 4-byte alignment and is four bytes in size because of the alignment restrictions introduced by using the “int” underlying data type for the bit field.

```
struct s {
     int bf : 5;
     char c;
};
```

This behavior is consistent with the other UNIX System V Release 4 ABIs.

2.2 Function Calling Conventions

2.2.1 Register Assignments

Table 2-3 shows the required register mapping for function calls. Some registers, such as the stack pointer, have specific purposes, while others are used for local variables, or to communicate function call arguments and return values.

Certain registers are bound to their purpose because specific instructions use them. For instance, subroutine call instructions write the return address into r15. The instructions used to save and restore registers on entry and exit from a function use r0 as a base register, making it most appropriate for the stack pointer register.

Refer to **2.2.3 Argument Passing** for an explanation of argument words and how they are allocated. Refer to **2.2.5 Return Values** for an explanation of the return buffer address.

Table 2-3 Register Assignments

Register Use Convention		
Name	**Usage**	**Cross-Call Status**
r0	Stack Pointer	Preserved
r1	Scratch	Destroyed
r2	Argument Word 1/Return Buffer Address	Destroyed/Preserved
r3	Argument Word 2	Destroyed
r4	Argument Word 3	Destroyed
r5	Argument Word 4	Destroyed
r6	Argument Word 5	Destroyed
r7	Argument Word 6	Destroyed
r8	Local	Preserved
r9	Local	Preserved
r10	Local	Preserved
r11	Local	Preserved
r12	Local	Preserved
r13	Local	Preserved
r14	Local	Preserved
r15	Link/Scratch	(Return Address)

2.2.1.1 Cross-Call Lifetimes

The 16 general-purpose registers are split between those preserved and those destroyed across function calls. This balances the need for callers to keep values in registers across calls against the need for simple leaf subroutines to perform operations without allocating stack space and saving registers. The preserved registers are called *non-volatile* registers. The registers that are destroyed are called *volatile* registers.

Registers r8 through r14 are preserved because the load and store multiple instructions deal with registers from N through 15. It is easy to save these registers in a single instruction.

The called subroutine can use any of the argument and scratch registers without concern for restoring their values. Preserved registers must be saved before being used and restored before returning to the caller. While the called function is not specifically required to save and restore r15, on entry r15 usually contains the return address, so the value must be preserved in order for execution to resume at the address of the instruction that follows the subroutine call.

Preserving r15 with the LDM and STM instructions is simple. Many implementations that save other non-volatile registers will also save r15. This is particularly useful when the called subroutine itself makes further subroutine calls.

The caller must preserve any essential data stored in argument and scratch registers. Data in these registers does not survive function calls. In particular, r1 is designated as a scratch register upon entry to a subroutine, and used to calculate stack frame adjustments for subroutines with large stack frames.

There is no register dedicated as a frame pointer. For non-alloca() functions, the frame pointer can always be expressed as an offset from the stack pointer. For alloca() functions and functions with very large frames, a frame pointer can be synthesized into one of the non-volatile registers.

Eliminating the dedicated frame pointer makes another register available for general use, with a corresponding improvement in generated code. This affects stack tracing for debugging. See **2.3 Runtime Debugging Support** for additional information.

2.2.2 Stack Frame Layout

The stack pointer points to the bottom (low address) of the stack frame. Space at lower addresses than the stack pointer is considered invalid and may actually be unaddressable. The stack pointer value must always be a multiple of eight.

Figure 2-1 shows typical stack frames for three functions, indicating the relative positions of local variables, parameters, and return addresses. The outbound argument overflow must be located at the bottom (low address) of the frame. Any incoming argument spill generated for *varargs* and *stdarg* processing must be at the top (high address) of the frame. Space allocated by **`Alloca()`** must reside between the outbound argument overflow and local variables areas.

The caller must stack argument variables that do not fit in the argument registers in the outbound argument overflow area. If all outbound arguments fit in registers, this area is not required. A caller may allocate argument overflow space sufficient for the worst-case call, use portions of it as necessary, and not change the stack pointer between calls.

The caller must reserve stack space for return variables that do not fit in the first two argument registers (e.g., structure returns). This return buffer area is typically located with the local variables. This space is typically allocated only in functions that make calls returning structures, and is not required.

The caller may stack the return address (r15) and the content of other local registers in the register save area upon entry to the called subroutine. If a called routine does not modify local variables (including r15), this area is not required.

Local variables that do not fit into the local registers are allocated space in the Local Variable area of the stack. If there are no such variables, this area is not required.

Beyond these requirements, a routine is free to manage its stack frame in any way desired.

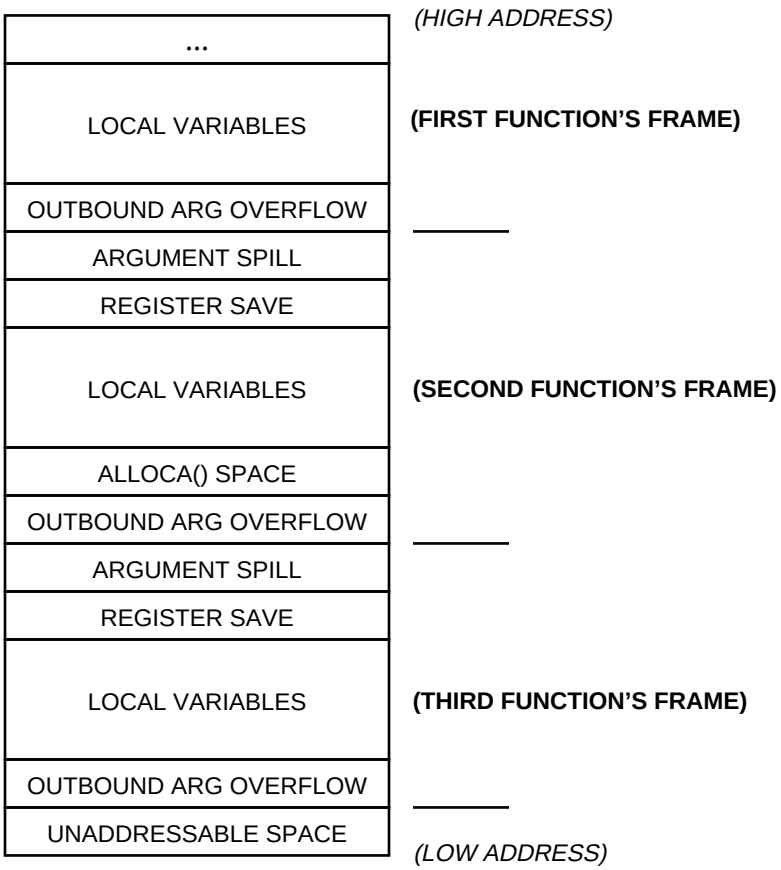

Figure 2-1 Stack Frame Layouts; First() calls Second() calls Third()

2.2.2.1 Extending the Stack

Stack maintenance is the responsibility of system software. In some environments, it may be valuable for compilers to probe the stack as they extend it in order to allow memory protection hardware to provide “guard pages”.

2.2.3 Argument Passing

The M•CORE uses six registers (r2–r7) to pass the first six words of arguments from the caller to the called routine. If additional argument space is required, the caller is responsible for allocating this space on the stack. This space (if needed by a particular caller) is typically allocated upon entry to a subroutine, reused for each of the calls made from that subroutine that have more arguments than fit into the six registers used for subroutine calls, and deallocated only at the caller’s exit point. All argument overflow allocation and deallocation is the responsibility of the caller.

At entry to a subroutine, the first word of any argument overflow can be found at the address contained in the stack pointer. Subsequent overflow words are located at successively larger addresses.

2.2.3.1 Scalar Arguments

Arguments are passed using registers r2 through r7, with no more than one argument assigned per register. Argument values that are smaller than a 32-bit register occupy a full register.

In addition, small argument values are right justified and possibly extended within the register. Small signed arguments (e.g., shorts) are sign extended; small unsigned arguments (e.g., unsigned shorts) are zero extended, while other small values (e.g., structures of less than four bytes) are not extended, leaving the upper bits of the register undefined. The caller is responsible for sign and zero extensions. Small arguments that are passed via the argument overflow mechanism are placed in the overflow word with the same orientation they would have if passed in a register; a char is passed in the low-order byte of an overflow word. Such small overflow arguments need not be sign extended within the argument word as they would be if passed in a register.

Arguments larger than a register must be assigned to multiple argument registers as long as there are argument registers available. Arguments that would be aligned on eight-byte boundaries in memory (double, long double, long long, or structures or unions containing a double, long double or long long) must begin in an even numbered register. Once all the argument registers are used, or if there are not enough registers left to hold a large argument, the argument and any subsequent arguments must be placed in the overflow area described above.

Large arguments must not be split when there are too few argument registers to hold the entire argument.

The caller is responsible for allocating argument overflow space and for deallocating any space needed for argument overflow. The only argument space that may be allocated or deallocated by the called routine is space used to place the register arguments in memory. This may be necessary for *stdargs* or structure parameters.

Alignment is forced for atomic data types; fundamental data types are not split.

2.2.3.2 Structure Arguments

Structures passed as arguments can be partially or wholly passed through the argument registers. A structure argument may overflow onto the stack only when all argument registers are full. In these cases, the caller must adjust the stack pointer to allocate the overflow area.

Structure arguments that are smaller than 32 bits have their value right justified within the argument register. The unused upper bits within the register are undefined.

Structure arguments larger than 32 bits are packed into consecutive registers. Structures that are not integral multiples of 32 bits in size have their final bits left justified within the appropriate register. This allows those bits to be stored with a 32-bit operation and be adjacent to the preceding portion of the structure.

2.2.4 Variable Arguments

The *stdarg* C macros provide a mechanism to handle variable length argument lists. The caller might not know whether the called function handles variable arguments, so the called routine is responsible for handling the nuances of variable argument lists.

2.2.4.1 Spilling Register Arguments

Variable argument lists are most easily handled by spilling one or more of the register arguments so that they are adjacent to any overflow arguments that are on the stack at function entry. The typical sequence should extend the stack several words, spill the argument registers after the last named argument into this space, and then proceed with the normal prologues to allocate a stack frame and save any non-volatile registers.

The *stdarg* macros can use the address of the first stored argument register for the *va_start* macro. The *va_arg* macro advances this pointer by an amount appropriate to the size of the type specified.

2.2.4.2 Legacy Code Compatibility

The M•CORE linkage convention provides a way for variable argument lists to be handled in a way that is compatible with legacy C code written for processors where the entire argument list is passed in memory.

The legacy behavior uses several more instructions, stack slots, and memory references than required by strict interpretation of the ANSI C standards. Tool generators must provide this legacy behavior as an option. It is not required as a default behavior.

To provide compatibility, the called function must spill all the argument registers, rather than just those beyond the registers that hold the named arguments. This is more pessimistic than required for the *stdarg* definitions, but provides the most compatibility.

Spilling is triggered for functions that take the address of any of their arguments.This allows non-standard *varargs* code (C code that works on processors with all arguments passed in memory) to run on the M•CORE.

The spilled arguments are a snapshot of their values at the time the function is entered. This requirement does not force the compiler to generate code that keeps the “live” value of the parameters in memory. For example, the following would not be required to print out the value “4”.

```
func(int a, int b, int c, ...)
{
     int *ip;
     use(c);
     ip = &b;
     ip++;
     *ip = 4;
     printf (“C now has value %d\n”, c);
}
```

The compiler is free to keep the value of *c* live in a register. The only requirement is to save a snapshot of the parameter passing registers (e.g., r2 through r7) during the function prologue.

2.2.5 Return Values

2.2.5.1 Scalar Values

Subroutines return values in the argument registers. Return values smaller than 32 bits occupy a full register. These must be right justified and zero or sign extended to 32 bits before return (refer to **2.2.3.1 Scalar Arguments**). Return values of 32 bits or fewer are returned in register r2.

Return values between 33 and 64 bits are returned in the register pair r2/r3. The portion of the data that would reside at a lower address if stored in memory is in r2. For example, r2 would contain the most significant 32 bits of the long long data type.

Return values larger than eight bytes are treated as structure return values and are returned through memory. The return value is placed in a caller-supplied buffer. The buffer address is passed from the caller to the called routine as a *hidden first argument* in register r2.

2.2.5.2 Structure Values

Structures can be returned in one of two ways.

Small structures (eight bytes or fewer) are returned in the register pair r2/r3. If the structure consists of four or fewer bytes, the value is returned in r2, right justified. This matches the way it would be justified when passed as an argument. If the structure consists of five to eight bytes, the first four bytes are returned in r2 and the trailing portion of the structure is returned left justified in r3.

This alignment is chosen to generate good code for code sequences such as

```
wom(..., bat(), ...)
```

where **`wom`** takes a structure argument of the same type returned by **`bat`**. The only work required is to perhaps change registers if the call to **`wom`** has the structure in some place other than r2/r3.

Structures larger than eight bytes are placed in a buffer provided by the caller. The caller must provide for a buffer of sufficient size; the buffer is typically allocated on the stack to provide re-entrancy and to avoid any race conditions where a static buffer may be overwritten. The address of the buffer is passed to the called function as a *hidden first argument* and arrives in register r2. The normal arguments start in register r3 instead of in r2, within the fundamental data type constraints.

The caller must provide this buffer for large structures even when the caller does not use the return value (e.g., the function was called to achieve a side-effect). The called routine can thus assume that the buffer pointer is valid and need not check the pointer value passed in r2.

When r2 is used to pass a buffer address, the called routine must preserve the value passed through r2. The caller can thus assume that r2 is preserved when the buffer address of a large structure is passed in r2. This is similar to the way in which *strcat* and *memcpy* return their respective destination addresses.

Often, the temporary buffer that is used for such structure returns is immediately used as a source for a *memcpy* to a final destination. For example, the sequence

```
struct s {...} s, sfunc();

s = sfunc();
```

will often be compiled with **`sfunc`** returning into a temporary buffer, which is immediately copied into **s**. Although the caller must know the address of the temporary buffer in order to provide it to the called routine, the address need not be recalculated. In turn, the called routine can use the address to copy the results into the temporary buffer using *memcpy*, which returns the destination address (e.g., r2 has the desired value), or passes it to in-line code which uses r2 as a base register.

2.3 Runtime Debugging Support

The most difficult aspect of M•CORE debugging is stack tracing. Tracing is complicated because the linkage convention does not mandate a frame pointer register and does not provide any back-chain construct. This section describes rules for generating function prologues that can be easily decoded by a debugger to determine the size of a stack frame, the location of the return address, and the location of any saved non-volatile registers.

2.3.1 Function Prologues

Function prologues acquire stack space needed by the function to store local variables. This includes space the function uses to save non-volatile registers. Prologue instruction sequences can take a number of forms. A set of working assumptions about function prologues follows.

- The function prologue is the only place in the function that acquires stack space, other than later calls to `alloca()`.
- The function prologue uses only the following classes of instructions.
  - — `subi r0,imm` (Note that this might appear multiple times in a prologue)
  - — `stm rn-r15,(r0)`
  - — `st.w rx,(r0,disp)`
  - — instructions that set and modify r1.
  - — *These are presumed to establish values for a relatively large frame. This sequence includes one of the following instructions:*
    - `lr.w r1, imm`
    - `movi r1, imm`
    - `bgeni r1, imm`
    - `bmaski r1, imm`

followed by zero or more of:

- `addi r1, imm`
- `subi r1, imm`
- `rsubi r1, imm`
- `not r1`
- `rotli r1, imm`
- `bseti r1, imm`
- `bclri r1, imm`
- `ixh r1,r1`
- `ixw r1,r1`

followed by:

- `sub r0, r1`

Whether `lrw` or the other sequence is used, the r1 value is subtracted from r0 to increase stack space. While this sequence is allowed to occur multiple times, code generators should generate a single literal of the appropriate value (e.g., summing two constants) rather than perform two subtractions.

— `mov rn,r0`

This is optional support for traceback through `alloca()`-using functions, and also marks the final instruction in the prologue.

- The function prologue is organized roughly as:
  - — If *stdarg*, acquire space to store volatile registers; store volatile registers.
  - — Acquire space to store non-volatile registers.
  - — Store non-volatile registers that may be modified in this function.
  - — Acquire any additional stack space required. This space acquisition might be folded in with earlier ones if the total space allocated is no more than 32 bytes.
  - — If needed in this function, copy the stack pointer into one of the non-volatile registers to act as a frame pointer.
  - — *Larger frames should allocate the register save space and then allocate the remainder of the required stack space rather than perform a single large stack acquisition. If the stack is acquired in a single allocation before the non-volatile registers are saved, then another base register is needed to reach the location for the stored registers. The prologue recognition code in the debugger does not recognize using alternate base registers to store the non-volatile registers as being part of the prologue.*
  - — This sequence allows the stack pointer to be modified several times.

2.3.2 Stack Tracing

Stack tracing for the M•CORE depends on the ability to determine the entry point for a function, given a PC value in that function. Since there are no unique prologue-only patterns in the instruction stream that can be identified by scanning backwards from the current PC, a symbol table for the executable file must be present. The symbols need not be complete DWARF information.

Placing a specific byte pattern just before the prologue is not sufficient to identify the beginning of a function because the pattern can also appear within the body of the function as part of a literal table. In code-size sensitive environments, the extra space consumed by such a byte pattern is undesirable.

The stack tracing code iteratively performs the following:

1. Get the current PC.

2. Find the beginning of the containing function. Stop if this can't be determined.

3. Decode the prologue starting at the function's entry.

4. Determine the "top of frame" from the framesize information described in the prologue. This is either an adjustment to the stack pointer or a "pseudo-frame pointer" if the prologue ends with a frame pointer generating instruction.

5. Recover stored non-volatile registers based on the offsets described in the prologue.

6. Repeat for the next frame.

SECTION 3 HIGH-LEVEL LANGUAGE ISSUES

3.1 C Preprocessor Predefines

All C language compilers must predefine the symbol __MCORE__ with the value “1” to indicate that the compiler targets the M•CORE processor. In the future, this value may be changed to correspond to different versions of the chip.

3.2 C In-Line Assembly Syntax

A C in-line assembly facility must be provided.

3.3 C Name Mapping

Externally visible names in the C language must be mapped through to assembly language without change.

For example, the following

```
void testfunc() { return;}
```

generates assembly code similar to the following fragment.

```
testfunc:
     jmp     r15
```

SECTION 4 OBJECT FILE FORMATS

M•CORE tools use ELF 2.0 object file formats and DWARF 1.1 debugging information formats, as described in *System V Application Binary Interface,* from The Santa Cruz Operation, Inc. ELF and DWARF provide a suitable basis for representing the information needed for embedded applications. This section describes particular fields in the ELF and DWARF formats that differ from the base standards for those formats.

4.1 Header Convention

The e_machine member of the ELF header contains the decimal value 39 (hexadecimal 0x27) which is defined as the name EM_MCORE.

Table 4-1 e_ident Field values

M•CORE e_ident Fields		
e_ident[EI_CLASS]	ELFCLASS32	For all 32-bit implementations
e_ident[EI_DATA]	ELFDATA2MSB	For all implementations

The ELF header e_flags member contains zero, because the M•CORE processor family defines no flags at this time.

4.2 Section Layout

4.2.1 Section Alignment

The object generator (compiler or assembler) provides alignment information for the linker. The default alignment is eight bytes. Object producers must ensure that generated objects specify required alignment. For example, an object file must reflect the fact that four-byte alignment is required in the data section.

4.2.2 Section Attributes

Table 4-2 defines section attributes that are available for M•CORE tools. These attributes are additions to the ELF standard flags shown in **Table 4-3**.

Table 4-2 M•CORE Section Attributes

M•CORE Section Attribute Flags	
Name	**Value**
SHF_MCORE_NOREAD	0x80000000

The SHF_MCORE_NOREAD attribute allows the specification of code that is executable but not readable. Plain ELF assumes that all segments have read attributes, which is why there is no read permission attribute in the ELF attribute list. In embedded applications, “execute-only” sections that allow hiding the implementation are often desirable.

Table 4-3 ELF Section Attributes

ELF Section Attribute Flags	
Name	**Value**
SHF_WRITE	0x00000001
SHF_ALLOC	0x00000002
SHF_EXECINSTR	0x00000004

4.2.3 Special Sections

Various sections hold program and control information. **Table 4-4** shows sections used by the system, the indicated types, and attributes. These are additions to ELF standards shown in **Table 4-5**. The ELF standard reserves section names beginning with a period (“.”), but applications may use those sections if their existing meanings are satisfactory.

M•CORE currently does not have a PIC standard. However, PIC section names are reserved for possible future implementation. The M•CORE is targeted at embedded systems. The linkage conventions for shared libraries are not necessary in this application space and are not defined as part of this ABI.

Table 4-4 M•CORE Tools Special Sections

M•CORE Reserved Section Names		
Name	**Type**	**Attributes**
.got	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR

NOTE

It is strongly recommended that read-only constants, such as string literals, be placed into the **`.rodata`** section instead of the **`.text`** section. The space that these add to **`.text`** can have a severe impact on addressability, requiring the use of larger branch instructions and reducing the chances for sharing of values in literal tables.

Table 4-5 ELF Sections

ELF Reserved Section Names		
Name	**Type**	**Attributes**
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	—
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	—
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.rel*	SHT_REL	—
.rela*	SHT_RELA	—
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	—
.symtab	SHT_SYMTAB	—
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

4.3 Symbol Table Format

There are no M•CORE symbol table requirements beyond the base ELF standards.

4.4 Relocation Information Format

4.4.1 Relocation Types

Relocation entries describe how to alter the instruction and data relocation fields as shown in **Table 4-6**. The choice of the relocation type numbers as encoded in the ELF object file is defined in **Table 4-7**.

Table 4-6 Relocation Types

word32	This specifies a 32-bit field occupying four bytes. This address is NOT required to be 4-byte aligned.
disp11	This corresponds to the scaled 11-bit displacement addressing mode. The relocation is the low-order 11 bits of the 16 bits addressed in the relocation type.
pcword32	This specifies a 32-bit field occupying four bytes. This address is NOT required to be 4-byte aligned.

The object file supports the 32-bit relocations for 32-bit data (addressing constants in memory). Both absolute and PC-relative relocations are defined.

Note that the 32 bits where the relocation is to be applied need not be on a 32-bit boundary. The relocation entry points to the address of the 32 bits to be adjusted by the relocation entry. The relocation adds the appropriate value (either the 32-bit value or the 32-bit displacement) to the existing contents of the 32 bits at that address.

A packed data structure can cause a 32-bit relocation to be misaligned in the object file. This might be done with a C compiler extension, or by means of hand-crafted assembly, in order to save data space (but the misaligned data must be accessed piece-wise to avoid alignment exceptions). The linker must be able to deal with this case.

Scaled 11-bit displacement mode is used in br, bf, bt, and bsr instructions. The 11-bit value indicates the number of halfwords from PC+2 to the target address. The relocation entry must point to the 16-bit instruction that contains the displacement.

The compiler or assembler must resolve displacement values for eight-bit indirect mode and four-bit negative displacement mode (used in the loopt instruction). The compiler must also fill in the eight-bit indirect mode to point to the appropriate literal table. Relocation entries are not expected in the object file.

4.4.2 Relocation Values

This section describes values and algorithms used for relocations. In particular, it describes values the compiler/assembler must leave in place and how the linker modifies those values.

Table 4-7 shows semantics of relocation operations. Key *S* indicates the final value assigned to the symbol referenced in the relocation record. Key *A* is the addend value specified in the relocation record. Key *P* indicates the address of the relocation (e.g., the address being modified).

Table 4-7 Relocation Type Encodings

Name	Value	Field	Calculation
R_MCORE_NONE	0	none	none
R_MCORE_ADDR32	1	word32	$S + A$
R_MCORE_PCRELIMM11BY2	3	disp11	$((S + A - P) \gg 1) \wedge 0x7ff$
R_MCORE_PCREL32	5	word32	$S + A - P$

4.4.2.1 32-Bit Relocations

Absolute 32-bit relocation adds the relocated symbols value to the existing content of the location specified. Consider the example

```
.long symbol+1234
```

The object file emitted by the compiler has a relocation entry for **`symbol`** that references the address of this word. The existing content of the 32 bits at the specified address are overwritten with the new value.

4.4.2.2 11-Bit Relocations

These relocations occur when br, bf, bt, and bsr instructions (typically bsr) reference a target that is not in the current object file. They can also occur when the target is in a separate section of the same object file, but these occurrences must be resolved by the compiler/assembler and not appear as relocation entries.

The relocation is calculated as shown in **Table 4-7**. The existing contents of the low-order 11 bits of the instruction are overwritten with the newly calculated displacement.

NOTE

The bsr instruction encoding is the distance from PC+2 to the target. This adjustment must be made in the compiler/assembler. The emitted relocation record for a bsr to symbol X must be to X+(–2); in other words, the symbol must be X and the addend field of the relocation record must contain –2.

4.5 Debugging Information Format

M•CORE tools must use DWARF 1.1 debugging information formats, as described in *System V Application Binary Interface,* from The Santa Cruz Operation, Inc.

Currently, no extensions to the DWARF standard are necessary to provide M•CORE debugging support. However, such extensions may be made in the future.

4.5.1 DWARF Register Numbers

DWARF generally describes the steps a debugger takes to locate variables in a program being debugged in machine-independent terms. However, the way in which the *OP_REG* and *OP_BASEREG* atoms are handled is machine-specific — these atoms require that a value (or the pointer to a value) be contained in a machine-specific register.

Table 4-8 shows the mapping between the values used in those atoms and the M•CORE register set. The entries for *r0* through *r15* specify the currently active set of general purpose registers; this is usually the primary register set. The entries for *r0'* through *r15'* specify the alternate register file. The control registers are encoded from 32 through 63.

Table 4-8 DWARF Register Atom Mapping for M•CORE

Atom	Register	Atom	Register	Atom	Register	Atom	Register
0	r0	1	r1	2	r2	3	r3
4	r4	5	r5	6	r6	7	r7
8	r8	9	r9	10	r10	11	r11
12	r12	13	r13	14	r14	15	r15
16	r0'	17	r1'	18	r2'	19	r3'
20	r4'	21	r5'	22	r6'	23	r7'
24	r8'	25	r9'	26	r10'	27	r11'
28	r12'	29	r13'	30	r14'	31	r15'
32	cr0	33	cr1	34	cr2	35	cr3
36	cr4	37	cr5	38	cr6	39	cr7
40	cr8	41	cr9	42	cr10	43	cr11
44	cr12	45	cr13	46	cr14	47	cr15
48	cr16	49	cr17	50	cr18	51	cr19
52	cr20	53	cr21	54	cr22	55	cr23
56	cr24	57	cr25	58	cr26	59	cr27
60	cr28	61	cr29	62	cr30	63	cr31
64	pc						

SECTION 5 LIBRARIES

The content of most libraries are platform and OS dependent. For this reason, they are beyond the scope of this document and are not addressed here. Some library functions are required to provide support for operations that are not supported directly by the M•CORE hardware. These library routines are specified in this section.

5.1 Compiler Assist Libraries

The M•CORE does not currently provide hardware support for floating point data types, nor for long long data types. Compilers should provide the functionality for some of these operations through the use of support library routines. The M•CORE Technology Center requires a single shared support library for all tool sets to eliminate redundant code.

The functions to be provided through support routines include:

- Floating point math routines
- Long long routines

Compilers that generate in-line code to provide these functions must make no references to the library functions.

Compilers that provide these functions by generating subroutine calls to the support libraries must use the standard interfaces.

In particular, it must be possible to link objects produced with different tool sets into single executables. This requires that:

- Compiler support library names not clash between tool sets
- Compiler support routines follow consistent linkage rules
- Linkers from different tool sets must either use the same support library names and interfaces, or must provide a mechanism to indicate where support libraries can be found.

Routines in the support libraries must satisfy the following constraints.

- The only external state information used is floating point rounding mode
- No global state can be modified
- Identical results must be returned when a routine is re-invoked with the same input arguments
- Multiple calls with the same input arguments can be collapsed into a single call with a cached result

These properties permit a compiler to make assumptions about variable lifetimes across library subroutine calls — values in memory won't change; previously de-referenced pointers need not be de-referenced again.

5.2 Floating Point Routines

These routines comply with ABI linkage conventions concerning registers that must be preserved across function calls. The routines have no side effects. They do not modify memory except as noted, thus allowing compilers to optimize de-referenced pointer values across calls. The routines always return the same value for the same inputs, allowing compilers to optimize subsequent calls away.

The data formats are as specified in IEEE 754. The routines are not required to compute results as specified in IEEE 754. Implementations of these routines must document the degree to which operations conform to the IEEE standard. Not all users of floating point require IEEE 754 precision and exception handling, and may not want to incur the overhead that complete conformance requires.

5.2.1 _d_add

```
double _d_add(double a, double b);
```

This function must return a + b computed to double precision.

5.2.2 int _d_cmp

```
int _d_cmp(double a, double b);
```

This function must perform an unordered comparison of the double precision values of a and b and must return an integer value that indicates their relative ordering:

Table 5-1 int_d_cmp Ordering Values

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

5.2.3 int _d_cmpe

```
int _d_cmpe(double a, double b);
```

This function must perform an ordered comparison of the double precision values of a and b and must return an integer value that indicates their relative ordering:

Table 5-2 int_d_cmpe Ordering Values

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2

5.2.4 double _d_div

```
double _d_div(double a, double b);
```

This function must return a / b computed to double precision.

5.2.5 float _d_dtof

```
float _d_dtof(double a);
```

This function must convert the double precision value of a to single precision and must return the single precision value.

5.2.6 int _d_dtoi

```
int _d_dtoi(double a);
```

This function must convert the double precision value of a to a signed integer by truncating any fractional part and must return the signed integer value.

5.2.7 long long _d_dtoll

```
long long _d_dtoll(double a);
```

This function must convert the double precision value of a to a signed long long by truncating any fractional part and must return the signed long long value.

5.2.8 unsigned int _d_dtou

```
unsigned int _d_dtou(double a);
```

This function must convert the double precision value of a to an unsigned integer by truncating any fractional part and must return the unsigned integer value.

5.2.9 unsigned long long _d_dtoull

```
unsigned long long _d_dtoull(double a);
```

This function must convert the double precision value of a to an unsigned long long by truncating any fractional part and must return the unsigned long long value.

5.2.10 int _d_feq

```
int _d_feq(double a, double b);
```

This function must perform an unordered comparison of the double precision values of a and b and must return one if they are equal, and zero otherwise.

5.2.11 int _d_fge

```
int _d_fge(double a, double b);
```

This function must perform an ordered comparison of the double precision values of a and b and must return one if a is greater than or equal to b, and zero otherwise.

5.2.12 int _d_fgt

```
int _d_fgt(double a, double b);
```

This function must perform an ordered comparison of the double precision values of a and b and must return one if a is greater than b, and zero otherwise.

5.2.13 int _d_fle

```
int _d_fle(double a, double b);
```

This function must perform an ordered comparison of the double precision values of a and b and must return one if a is less than or equal to b, and zero otherwise.

5.2.14 int _d_flt

```
int _d_flt(double a, double b);
```

This function must perform an ordered comparison of the double precision values of a and b and must return one if a is less than b, and zero otherwise.

5.2.15 int _d_fne

```
int _d_fne(double a, double b);
```

This function must perform an unordered comparison of the double precision values of a and b and must return one if they are unordered or not equal, and zero otherwise.

5.2.16 double _d_itod

```
double _d_itod(int a);
```

This function must convert the signed integer value of a to double precision and must return the double precision value.

5.2.17 double _d_lltod

```
double _d_lltod(long long a);
```

This function must convert the signed long long value of a to double precision and must return the double precision value.

5.2.18 double _d_mul

```
double _d_mul(double a, double b);
```

This function must return a * b computed to double precision.

5.2.19 double _d_neg

```
double _d_neg(double a);
```

This function must return –a.

5.2.20 double _d_sub

```
double _d_sub(double a, double b);
```

This function must return a – b computed to double precision.

5.2.21 double _d_ulltod

```
double _d_ulltod(unsigned long long a);
```

This function must convert the unsigned long long value of a to double precision and must return the double precision value.

5.2.22 double _d_utod

```
double _d_utod(unsigned int a);
```

This function must convert the unsigned integer value of a to double precision and must return the double precision value.

5.2.23 int _fp_round

```
int _fp_round(int rounding_mode);
```

This function must set the rounding mode for sfpe library routines. **Table 5-3** shows the mode selected by different argument values. This function must return the resulting rounding mode (zero for round to nearest, etc.) if that rounding mode is supported by the sfpe library routines. Only round to nearest (function returns zero) is required for conformance. The rounding mode may be stored in memory.

Table 5-3 int _fp_round Values

Rounding_mode Value	Rounding Operation
0	Round toward nearest
1	Round toward zero
2	Round toward positive infinity
3	Round toward negative infinity

5.2.24 float _f_add

```
float _f_add(float a, float b);
```

This function must return a + b computed to single precision.

5.2.25 int _f_cmp

```
int _f_cmp(float a, float b);
```

This function must perform an unordered comparison of the single precision values of a and b and must return an integer value that indicates their relative ordering:

Table 5-4 int_f_cmp Ordering Values

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

5.2.26 int _f_cmpe

```
int _f_cmpe(float a, float b);
```

This function must perform an ordered comparison of the single precision values of a and b and must return an integer value that indicates their relative ordering:

Table 5-5 int_f_cmpe Ordering Values

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2

5.2.27 float _f_div

```
float _f_div(float a, float b);
```

This function must return a / b computed to single precision.

5.2.28 int _f_feq

```
int _f_feq(float a, float b);
```

This function must perform an unordered comparison of the single precision values of a and b and must return one if they are equal, and zero otherwise.

5.2.29 int _f_fge

```
int _f_fge(float a, float b);
```

This function must perform an ordered comparison of the single precision values of a and b and must return one if a is greater than or equal to b, and zero otherwise.

5.2.30 int _f_fgt

```
inf _f_fgt(float a, float b);
```

This function must perform an ordered comparison of the single precision values of a and b and must return one if a is greater than b, and zero otherwise.

5.2.31 int _f_fle

```
int _f_fle(float a, float b);
```

This function must perform an ordered comparison of the single precision values of a and b and must return one if a is less than or equal to b, and zero otherwise.

5.2.32 int _f_flt

```
int _f_flt(float a, float b);
```

This function must perform an ordered comparison of the single precision values of a and b and must return one if a is less than b, and zero otherwise.

5.2.33 int _f_fne

```
int _f_fne(float a, float b);
```

This function must perform an unordered comparison of the single precision values of a and b and must return one if they are unordered or not equal, and zero otherwise.

5.2.34 double _f_ftod

```
double _f_ftod(float a);
```

This function must convert the single precision value of a to double precision and must return the double precision value.

5.2.35 int _f_ftoi

```
int _f_ftoi(float a);
```

This function must convert the single precision value of a to a signed integer by truncating any fractional part and must return the signed integer value.

5.2.36 long long _f_ftoll

```
long long _f_ftoll(float a);
```

This function must convert the single precision value of a to a signed long long by truncating any fractional part and must return the signed long long value.

5.2.37 unsigned int _f_ftou

```
unsigned int _f_ftou(float a);
```

This function must convert the single precision value of a to an unsigned integer by truncating any fractional part and must return the unsigned integer value.

5.2.38 unsigned long long _f_ftoull

```
unsigned long long _f_ftoull(float a);
```

This function must convert the single precision value of a to an unsigned long long by truncating any fractional part and must return the unsigned long long value.

5.2.39 float _f_itof

```
float _f_itof(int a);
```

This function must convert the signed integer value of a to single precision and must return the single precision value.

5.2.40 float _f_lltof

```
float _f_lltof(long long a);
```

This function must convert the signed long long value of a to single precision and must return the single precision value.

5.2.41 float _f_mul

```
float _f_mul(float a, float b);
```

This function must return a * b computed to single precision.

5.2.42 float _f_neg

```
float _f_neg(float a);
```

This function must return –a.

5.2.43 float _f_sub

```
float _f_sub(float a, float b);
```

This function must return a – b computed to single precision.

5.2.44 float _f_utof

```
float _f_utof(unsigned int a);
```

This function must convert the unsigned integer value of a to single precision and must return the single precision value.

5.2.45 float _f_ulltof

```
float _f_ulltof(unsigned long long a);
```

This function must convert the unsigned long long value of a to single precision and must return the single precision value.

5.3 Long Long Integer Routines

These routines comply with ABI linkage conventions concerning registers that must be preserved across function calls. The routines have no side effects. They do not modify memory except as noted, and thus allow compilers to optimize de-referenced pointer values across calls. The routines always return the same value for the same inputs, allowing compilers to optimize subsequent calls away.

5.3.1 long long __div64

```
long long __div64( long long a, long long b )
```

This function computes the quotient a / b, truncating any fractional part, and returns the signed long long result.

If the divisor has the value zero, the behavior is undefined.

5.3.2 long long __mul64

```
long long __mul64(long long a, long long b);
```

This function computes the product a * b and returns the result.

5.3.3 long long __rem64

```
long long __rem64( long long a, long long b )
```

This function computes the remainder upon dividing a by b and returns the signed long long result.

If the divisor has the value zero, the behavior is undefined.

5.3.4 unsigned long long __udiv64

```
unsigned long long __udiv64( unsigned long long a, unsigned long long b )
```

This function computes the quotient a / b, truncating any fractional part, and returns the unsigned long long result.

If the divisor has the value zero, the behavior is undefined.

5.3.5 unsigned long long __urem64

```
unsigned long long __urem64( unsigned long long a, unsigned long long b )
```

This function computes the remainder upon dividing a by b and returns the unsigned long long result.

If the divisor has the value zero, the behavior is undefined.

SECTION 6 ASSEMBLER SYNTAX AND DIRECTIVES

6.1 Sections

The output of the assembler consists of, in part, sections whose content is determined by the assembler input. Sections containing code are aligned to 2-byte boundaries. Sections containing data are aligned so that the alignment requirements of the data contained in the section is preserved.

6.2 Input Line Lengths

The assembler may limit input lines, but such a limit must be at least 2100 characters in length. This gives the ability to construct an expression containing a symbol of maximum supported length (2048 bytes) and a data-allocation pseudo-instruction. For example:

```
.long     longsymbol
```

The assembler is allowed to support longer lines. If the assembler imposes a limit on the length of an input line, the assembler must issue a diagnostic if that limit is exceeded.

6.3 Syntax

An assembler source file contains a list of one or more assembler statements. Each statement is terminated with a newline character or a "`;`" character. The "`;`" character does not terminate the statement if it appears within a string literal or inside a comment. Empty statements (i.e. blank lines) are ignored.

Each statement consists of zero or more labels, at most one mnemonic, with the remainder of the statement being arguments specific to the mnemonic.

Labels are symbols that are followed by a "`:`". Temporary labels are allowed and are indicated by a non-zero digit (1–9) instead of a symbol. Duplicate temporary labels are allowed and references to them are resolved by searching for the nearest source line with the label. References to temporary labels must have a "b" or "f" suffix appended to the digit to indicate which direction to search.

Labels that begin with "`.`" (period) are considered local labels. The assembler does not include these symbols in the symbol table of the generated object file.

Mnemonics fall into three categories: instructions, pseudo-instructions, and directives. Instruction mnemonics map one-to-one into an M•CORE opcode. Pseudo-instructions map into sequences of M•CORE opcodes. Directives always start with a "`.`" and are used to control the assembly and allocate data areas. All mnemonics are case sensitive and must be specified in lower case.

White space in assembler source files is ignored except as a separator between mnemonics and when embedded within string literals or character constants. Multiple white space characters are functionally equivalent to a single white space character except within literals and character constants.

Comments in assembler source are indicated by the following:

- A "//" sequence indicates a comment reaching to the end of the line.
- A "#" character, when not part of a valid preprocessing directive, indicates a comment reaching to the end of the line.

Comments are terminated only by the end of the line. The ";" character does not terminate a comment. A multi-line comment, e.g. "/* */", is not supported since most assemblers are inherently line oriented.

Comments can never begin or end within a string literal or character constant.

6.3.1 Preprocessing

The assembler is not required to provide macro preprocessing. This functionality can be provided by existing preprocessors that conform to the ANSI standard. If the assembler does provide preprocessing, then it must conform to the "C" language preprocessing standard and the following paragraph does not apply.

An assembler command line option will enable the following behavior. Any line with a "#" character in the first column is assumed to be line and file information from the preprocessor. The assembler must use this information in error messages. This allows a programmer to relate an error back to the line and file of the original source file before preprocessing. The file and line information from the preprocessor is in the form:

```
# number "filename"
```

Any other preprocessor lines that do not match this form are ignored by treating them as comments.

6.3.2 Symbols

Symbols must begin with a character in the set: a–z, A–Z, . (period), or _ (underscore). The remaining characters in a symbol may be in that set plus the digits 0–9. Symbols are case sensitive and all characters in the symbol are significant. Symbols may be limited in length but that limit must be at least 2048 characters. If there is a limit on symbol length, symbols that exceed the limit must cause an error message to be emitted.

Silent truncation of long symbols is undesirable. This is intended to avoid silent errors where two long symbols differ only at some point after the tools have stopped keeping track of significant characters. The "$" character is not allowed in a symbol name because it is not a universally supported character on non-U.S. keyboards.

The special symbols created by temporary labels can only be referenced within a single source file. These references must consist of a single digit followed by a “b” or “f” to indicate the direction of the nearest matching label.

The “.” symbol will always indicate the current location within the current section at the start of the current statement. Thus:

```
movi       r3,15;br.
br         .
```

results in three instructions, two of which branch to themselves.

The “.” symbol is used instead of “*” because it avoids conflicts with “*” as a multiply operator.

6.3.3 Constants

The same constants and lexical expression of constants that are available in C are allowed in the assembly. This includes hex, octal, decimal, float, double, character, and strings. Both character and string constants have characters, ‘ and “ respectively, to delimit them. Multiple characters within character constant are each treated like a base 256 number. e.g. ‘1234’ equals 0x31323334.

The syntax of constants is chosen to be familiar to C programmers. The use of special characters in the syntax for constants must be avoided as they are used in expressions. In addition, the “$” character is not a universally supported character on non-U.S. keyboards.

6.3.4 Expressions

Addition, subtraction, multiplication, division, modulus, logical anding, inclusive oring, exclusive oring, negating, complementing, and shifting operations are supported by the assembler for the generation of constants or relocatable expressions in the argument portion of a statement. These operations have the semantics and precedence of their equivalent C language operations. Parenthesis can be used to force particular bindings of operations. All operations are done as if on 32-bit unsigned values. The syntax of expressions is chosen to be familiar to C programmers.

Expressions can involve more than one relocatable value as long as the assembler can resolve the expression to remove all or all but one of the relocatable values. For example, the difference between two labels in the same section reduces to an assemble time constant.

Relocatable expressions must evaluate down to a possibly-zero offset from a relocatable address. The linker is not required to provide the ability to store the value “5 times the value of this relocatable symbol”.

6.3.5 Operators and Precedence

Table 6-1 shows the operators available to the assembly programmer. The table is arranged in order of precedence; the higher precedence operators appear earlier in the table. These are the same operators used in the C language.

Table 6-1 Assembly Expression Operators

Assembly Expression Operators		Precedence
-	unary negation	1
~	unary logical complement	
*	multiplication	2
/	division	
%	modulus	
+	addition	3
-	subtraction	
<<	left shift	4
>>	right shift	
&	logical and	5
^	logical exclusive or	6
|	logical inclusive or	7

Operations may be grouped with parentheses to force a particular precedence.

6.3.6 Instruction Mnemonics

The instruction opcode mnemonics are listed in the *M•CORE Reference Manual* (MCORERM/AD).

6.3.7 Instruction Arguments

Register arguments within the argument portion of a statement are indicated by the character, "`r`" or "`R`" followed by the register number (0 through 15). Register 0 (`r0`) can also be specified as "`sp`".

Instructions that use the PC relative indirect addressing (lrw, jsri, jmpi) take two argument syntaxes. The first syntax is of the form:

```
lrw        r0,0x12345678
lrw        r1,0x4321
lrw        r2,0x4321
lrw        r3,0x4321
```

The assembler collects these argument values into a literal table, possibly allowing several instructions to reuse the same slot, and emit them at an appropriate point in the output. Such a point may be after the nearest unconditional branch. In some situations, such a location might not arise before the span of the lrw/jsri/jmpi instruction is exhausted. In such cases, the assembler must spill the literal table before the span is exhausted and provide a branch around the literal table.

The assembler provides a mechanism that allows the user to force a dump of the currently outstanding literals by using the **`.literals`** pseudo-instruction. Any literals that have not yet been emitted are emitted when this directive is encountered. When the assembler input is exhausted, the assembler emits any literals that have not yet been emitted, as if a **`.literals`** pseudo-instruction was appended to the assembly source.

NOTE

The assembler is allowed, but not required, to attempt to optimize code size by doing "optimal" literal placement. This interacts with the expansion of **`jbt`** and **`jbf`** pseudo-operations. Also, if literals must be output after an instruction that is not an unconditional transfer of control, the assembler must insure that a branch around the literal table is also generated.

The second form uses a `[label]` notation for the literal. In this case, the supplied argument is the label of the address containing the value to be loaded. This gives the assembler programmer complete control over the placement and sharing of literals.

```
        lrw     r0,[lit0]
        lrw     r1,[lit1]
        lrw     r2,[lit1]
        lrw     r3,[lit1]
        ...
.align 2
lit0:   .long   0x12345678
lit1:   .long   0x4321
```

NOTE

The user is responsible for insuring that the specified label is 4-byte aligned when using the `[label]` literal syntax.

The M•CORE instruction set does not directly support position independent code so it is up to the assembler programmer or compiler to synthesize PC-relative branches and subroutine calls. To help support this, a 32-bit PC relative argument type is allowed and is indicated by an expression that is evaluated as a delta from "`.`". Any symbols in the expression must be within the same section as the instruction so the assembler can resolve it to a constant offset. This can be done in the following manner (assuming r1 and r15 are available):

```
        bsr     .+2
        lrw     r1,symbol-.
        add     r1,r15
        jsr     r1
        ...
symbol: subi    r0,12
        ...
```

6.4 Assembler Directives

Assembler directives are used to control the assembly of the source code as well as reserving and/or initializing areas for data. All assembler directive mnemonics begin with a "`.`".

Only the `.align`, `.comm`, and `.lcomm` directives align the location counter to a known boundary. All other mnemonics, including `.long`, do not imply alignment. It is up to the assembler programmer or compiler to explicitly align these locations to avoid runtime misalignment faults. For operations that specify alignment values (e.g., `.align`, `.comm`, and `.lcomm`), the value specified is log_2 of the alignment. For example, the value "3" specifies 8-byte alignment.

All data values emitted by assembler directives will be in big-endian order.

This alignment behavior is needed to support packed data structures. Packed data structures explicitly allow misaligned fundamental types to save data space at the expense of additional code to pack and unpack the structures. Note that the ABI does not specify how a user expresses such misaligned references at the C source level.

The directive syntax in this manual uses "[" and "]" to indicate an optional field. The "{" and "}" syntax indicates zero or more repetitions of a field.

6.4.1 .align abs-exp [, abs-exp]

Aligns the location counter to the boundary indicated by the first constant expression. The integral alignment argument is log_2 of the alignment, e.g. the value "3" specifies 8-byte alignment. Negative alignment values are treated as zero, indicating 1-byte alignment.

The second, optional expression is the value to be filled into the bytes between the old location and new location. If unspecified, the bytes will be filled with zeros.

NOTE

The maximum alignment allowed is not constrained by the assembler. But in order for the assembler to be able to resolve expressions between symbols in the section, the linker must guarantee that the resulting section will be aligned to the largest alignment required within the section. This can be true for every loadable section from every source file, so large alignments should be used conservatively to avoid large gaps in the final load image.

6.4.2 .ascii "string" {, "string"}

Reserves and initializes space for one or more strings given. Each assembled string will not be null-terminated and will fill consecutive addresses. No alignment is implied.

6.4.3 .asciz "string" {, "string"}

Same as .ascii except the strings will be null terminated.

6.4.4 .byte exp {, exp}

Assembles consecutive bytes with the one or more values given by the expression(s). No alignment is implied.

Values larger than eight bits are truncated to fit into eight bits. This also generates a warning diagnostic.

6.4.5 .comm symbol, length [, align]

Declares an area of length bytes in the `.bss` section that will be shared by different files. If another file declares a longer length, then the length will be the maximum of all the declared lengths.

The alignment, if specified, is log_2 of the alignment. The value "3" specifies 8-byte alignment. The units are the same as in the `.align` directive. If no alignment is specified, the assembler will naturally align the symbol according to the largest natural type that can be contained in an entity of that size. Entities of eight bytes and larger are 8-byte aligned, entities of four bytes are 4-byte aligned, entities of two and three bytes are 2-byte aligned, single-byte entities are 1-byte aligned.

6.4.6 .data

Equivalent to:

```
.section  .data,"RW"
```

6.4.7 .double float {, float}

Assembles floating point values into IEEE 64-bit floating point numbers. The numbers will be consecutive and no alignment is implied.

6.4.8 .equ symbol, expression

Sets the value of the symbol to the expression. If the expression value cannot be resolved to an absolute or relocatable value after all assembler passes are complete, the assembly will be aborted with an error.

6.4.9 .export symbol {, symbol}

Causes the symbol to appear in the emitted symbol table in the resulting object file. The symbol may be defined within the file or it may be defined within an external file.

6.4.10 .fill count [, size [, value]]

Emits count copies of the value given. Only the least significant `size` bytes of `value` are replicated. The size must be a value ranging from one through eight; the default size is one byte. The default value is zero.

All three arguments are integral absolute expressions.

6.4.11 .float float {, float}

Assembles floating point values into IEEE 32-bit floating point numbers. The numbers will be consecutive and no alignment is implied.

6.4.12 .ident "string"

Places the string in the `.comment` section of the object file reserved for identification purposes. This is used for version tracking and source-to-binary audit trails.

6.4.13 .import symbol {, symbol}

Indicates that the symbols are defined externally from this file. All undefined symbols that are not declared as imported will cause a warning message to be issued by the assembler. Symbols that have been declared external but are not referenced should not appear in the symbol table of the emitted object file.

6.4.14 .literals

Causes the assembler's accumulated literal table for the `jmpi`, `jsri`, and `lrw` instructions for the current section to be emitted. Can be used by the assembler programmer to flush literal tables at the exact point desired.

6.4.15 .lcomm symbol, length [, alignment]

Reserve length bytes for a named local common area in the `.bss` section. The allocations of symbols in the `.bss` section will be in the same order as the `.lcomm` statements in the source file.

NOTE

Preserving the allocation order allows the compiler to use fixed offsets from a `bss` pointer to access several related variables.

The optional alignment value is log_2 of the desired alignment; a value of "3" specifies eight byte alignment. If no alignment is specified, the assembler will naturally align the symbol according to the largest natural type that can be contained in an entity of that size. Entities of eight bytes and larger are 8-byte aligned, entities of four bytes are 4-byte aligned, entities of two and three bytes are 2-byte aligned, single-byte entities are 1-byte aligned.

6.4.16 .long exp {, exp}

Emits four byte values consecutively.

6.4.17 .section name [, "attributes"]

Assemble subsequent statements onto the end of the named section.

Section names obey the same syntax as symbol names.

The attributes supported are the access permissions (read, write, and execute) and the allocation bits (yes or no). Permissions and allocation are indicated by any combination of the letters `RWXANrwxan` with no separators between them. The attributes are specified as a quoted string. The attribute characters are explained in **Table 6-2**.

Table 6-2 M•CORE Section Attribute Encodings

Section Attribute Encodings	
R or r	Section is to be readable.
W or w	Section is to be writable.
X or x	Section contains executable code.
A or a	Section is to be allocated space in the loaded image.
N or n	Section is NOT to be allocate space in the loaded image.

A missing attribute list indicates that the section should have all permissions (RWX) and address space will be allocated in the load map. An empty attribute list (e.g., an empty quoted string) specifies an allocated but inaccessible section.

A missing attribute list generates the default permissions.

Multiple specifications of a section take the attributes from the first specification of the section.

```
.sectionsectionname,"RX"
.sectionsectionname,"RW"
```

The `RW` attribute is ignored and the section `sectionname` will have read and execute permissions.

6.4.18 .short exp {, exp}

Emits two byte values consecutively.

6.4.19 .text

Equivalent to:

```
.section.text,"RX"
```

6.4.20 .weak symbol [, symbol]

Specify a weak external symbol definition. If *symbol* is not otherwise defined at link time, it has the value zero. Multiple symbols can be specified on the same line.

6.5 Pseudo-Instructions

The assembler also supports several pseudo-instructions which are expanded into one or more machine instructions.

Some pseudo-instructions are used to delay selection of instructions until relative addresses are resolved. For example, a smaller relative branch instruction could be emitted instead of a larger absolute jump instruction if the decision is delayed until the branch distance is known.

Other pseudo-instructions are for the assembler programmers convenience. For example, the “clear the condition bit” (`clrc`) instruction is another mnemonic for a compare of r0 being not equal to r0. Also, the mnemonics for the load/store instructions (ldb, ldh, ldw, stb, sth, stw) have alternate forms (ld.b, ld.h, ld.w, st.b, st.h, st.w).

6.5.1 clrc

Clear the condition code bit (C) in the status register. Emits the opcode equivalent to:

```
cmpner0,r0
```

6.5.2 cmplei rd, n

Perform a signed comparison of the value in *rd* with the constant *n*. *N* is allowed to have the values 0 through 31. Emits the opcode equivalent to:

```
cmpltird, n+1
```

6.5.3 cmpls rd, rs

Compare if the unsigned value in *rd* is lower or the same as the unsigned value in *rs*. Emits the opcode equivalent to:

```
cmphsrs, rd
```

6.5.4 cmpgt rd, rs

Compare if the signed value in *rd* is greater than the signed value in *rs*. Emits the opcode equivalent to:

```
cmpltrs, rd
```

6.5.5 jbsr label

Call the subroutine identified by *label*. Use the relative branch to subroutine instruction if the subroutine is within range, otherwise use an absolute jump to subroutine. Emits one of the following sequences:

```
bsrlabel
```

Or:

```
jsrilabel
```

6.5.6 jbr label

Continue execution at the instruction identified by *label*. Use the relative branch instruction if the label is within range, otherwise use an absolute jump to the label. Emits the equivalent of one of the following sequences based on the distance to the target label.

```
br label
```

Or:

```
jmpilabel
```

6.5.7 jbf label

Continue execution at the instruction identified by *label* only if the condition code bit is false. Use the relative conditional branch instruction if the label is within range, otherwise use a conditional branch around an absolute jump to the label. Emits the equivalent of one of the following sequences based on the distance to the target label.

```
        bf label
```

Or:

```
        bt 1f
        jmpilabel
    1: ...
```

The temporary label “1” is here for illustration purposes; it is not emitted. The expansion of jbf will not cause a problem for the following fragment.

```
        bt 1
        ...
        jbflabel
    1: ...
```

6.5.8 jbt label

Continue execution at the instruction identified by *label* only if the condition code bit is true. Use the relative conditional branch instruction if the label is within range, otherwise use a conditional branch around an absolute jump to the label. Emits the equivalent of one of the following sequences based on the distance to the target label.

```
        bt label
```

Or:

```
        bf 1f
        jmpilabel
    1: ...
```

The temporary label “1” is here for illustration purposes; it is not emitted. The expansion of jbt will not cause a problem for the following fragment.

```
        bt 1
        ...
        jbtlabel
    1: ...
```

6.5.9 neg rd

Negates the value in rd. Emits the opcode equivalent to:

```
        rsubird,0
```

6.5.10 rotlc rd, 1

Rotates the value in rd left by one bit. The carry bit is rotated into least significant bit while the most significant bit that was rotated out is saved in the carry bit. Emits the opcode equivalent to:

```
addcrd,rd
```

6.5.11 rotri rd, imm

Rotates the value in rd right by the number of bits specified in imm. Emits the opcode equivalent to:

```
rotli rd,32-imm
```

An immediate value of 0 is not allowed.

6.5.12 rts

Return from subroutine. Emits the opcode equivalent to:

```
jmpr15
```

6.5.13 setc

Set the condition code bit (C) in the status register. Emits the opcode equivalent to:

```
cmphsr0,r0
```

6.5.14 tstle rd

Test for a negative or zero value in the specified register. Emits the opcode equivalent to:

```
cmpltird,1
```

6.5.15 tstlt rd

Test for a negative value in the specified register. Emits the opcode equivalent to:

```
btstird,31
```

6.5.16 tstne rd

Test for a non-zero value in the specified register. Emits the opcode equivalent to:

```
cmpneird,0
```

INDEX

RECORD OF CHANGES

Revision	Date	Description
Original	03 OCT 97	Publication based on Motorola design specifications