



Microengine Version 2 (MeV2)

Microengine C Compiler Coding Considerations

Version 0.4
June 25, 2003

Revision History

Rev.	Date	Reason for Changes
0.1	03/08/2003	Initial Draft
0.2	04/30/2003	Feedback/clarifications from engineering + addition of new material
0.3	06/06/2003	Fixing of typos and added clarifications after a formal peer review
0.4	06/25/2003	Retitled document for SDK releases.

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's Web site at <http://www.intel.com>.

Copyright © Intel Corporation, 2003

Intel is a registered trademark and XScale is a trademark of Intel Corporation or its subsidiaries in the United States and other countries

*Other names and brands may be claimed as the property of others.

Abstract

Intel provides an integrated tool suite to developers to allow them to design, develop, test and deploy applications based on the Intel network processors. This tool suite consists primarily of an assembler/compiler toolchain for the Microengine Version 2 (MEv2) target, a cycle-accurate simulation environment, and a programming framework for writing portable and reusable code.

The microengine target of an Intel network processor has a non-traditional, highly parallel architecture. It is so designed to address the unique challenges of network processing at high speeds. The operating environment is such that application performance is obtained by a complex balance of tradeoffs: sequential vs. parallel processing, memory and I/O latency hiding, and communication bus load balancing.

As such, the Microengine C compiler (also referred to as “the compiler” in this document) has special considerations for those who would program with it versus an ANSI C compiler designed for a general-purpose processor. This document outlines many software architecture and code performance topics under this heading. Some of the sections describe pitfalls - resulting in incorrect functioning of the code - for a new Microengine C programmer, while others hold tips for improving compiler optimization. Finally, this document includes discussion on best known methods for Microengine C debugging, such as understanding the live range information provided by the compiler and the run-time tools provided by the Developer’s Workbench.

Table of Contents

Revision History	ii
Abstract	iii
Table of Contents	iv
Overview	5
Purpose and Scope of this Document	5
Definitions/Acronyms.....	6
Related Documents.....	6
Software Implementation Considerations.....	7
Variable Live Range Analysis.....	7
Asynchronous I/O operations	8
Referring to a variable indirectly	9
Memory Intrinsic with Variable ref_cnt.....	11
Live Range Summary.....	13
Data Alignment.....	13
Packing Structures	14
Overriding Natural Alignment	16
Efficient Structure Access	17
Sizing of Structure Members	17
The Use of Unions.....	19
Miscellaneous Considerations.....	20
Signed vs. Unsigned Integers.....	21
Tuning Established Code	22
Register Spillage	22
Handling Register Spillage	22
Functions.....	23
Inlining Functions	23
Optimizing Pointer Arguments.....	24
Miscellaneous Optimizations	25
Conditional Statements	25
Compiler Defer Slot Filling.....	26
Debugging Techniques	27
Compile-time Information	27
Performance Information.....	27
Run-Time Debug Tools	30
Summary.....	30

Overview

Purpose and Scope of this Document

The purpose of this document is to get new Microengine C programmers writing correct, functional and optimized code quickly. To do so, the document will cover three main topics. First, it will highlight the potential pitfalls for ANSI C programmers new to programming network processors. These are topics for which unaware programmers will unwittingly prompt the compiler to generate incorrect code in relation to their actual intent. Second, the document will discuss tips to help the programmer elicit the best possible code from the compiler. This class of topics will be for the programmer to keep in mind when coding, though not implementing them will not result incorrect code, simply suboptimal. And finally, the document will describe the compile-time and run-time features of the network processor development tools invaluable for a programmer to debug his or her Microengine code. These features are part of both the compiler and the Developer's Workbench GUI.

It should be noted that readers of this document should already have a strong understanding of the network processor architecture as well as a basic knowledge of Microengine C syntax. If not, please consult the latest appropriate Hardware Reference Manuals and Programmer Reference Manuals for the former and the Microengine C Language Support Reference Manual for the latter.

Definitions/Acronyms

Term	Definition
U32	“unsigned int”

Related Documents

Title	Location
Microengine C Compiler Language Support Reference Manual	IXA SDK 3.1
IXP2800 Hardware Reference Manual	IXA SDK 3.1
IXP2400 Hardware Reference Manual	IXA SDK 3.1
IXP2400/IXP2800 Programmer’s Reference Manual	IXA SDK 3.1

Software Implementation Considerations

The Microengine C compiler provides a high-level-language programming environment for the network processors to reduce application development time and reduce the need for specialized knowledge. That said, there are specific considerations a Microengine C programmer needs to be made aware of to program with confidence in terms of correct Microengine behavior and performance. In fact, many of the issues in this section do not exist the compiler of a general-purpose processor and therefore might not be obvious on first look.

Variable Live Range Analysis

Register allocation and other compiler optimizations depend on having correct live range information for register variables. A live range of a register variable is the period between the definition of this variable and the last use of the defined value. When a register variable has multiple definitions in the program and each definition has sequential reads, multiple live ranges are assigned to the same variable. It follows that multiple reads in the middle of live range are fine, but a write into the same variable in the middle of a live range will split it.

The compiler automatically calculates the live range of a register variable through code analysis with the fact that a live range always starts with a write into the variable and terminates at the point where there is no subsequent read of this written value (i.e. the last read point). A register variable has the same physical register assigned to it for the span of one live range; however, it could have different physical registers assigned to it across different live ranges.

(For more information on how the compiler calculates variable live ranges, please see the Microengine C Compiler Support Reference Manual.)

There are times, however, where the compiler will not be able to calculate the live range of a variable correctly. Specifically, the programmer will have to intervene when a variable is implicitly read or written at a point in the code where the variable is not referred to by name. For example, some asynchronous memory operations or event signaling can be done in such a manner. In these cases, the compiler has no way to figure out the true start or end of the live range through code inspection. This situation can lead to suboptimal register allocation, or worse, incorrect code generation.

There are some major code constructs that merit the use of the `__implicit_read()` and `__implicit_write()` intrinsic functions to help the compiler with live range analysis. These intrinsics do not generate any new code per se, but rather, allow the programmer to manually extend or shorten the live range of a variable by providing a clue to the compiler as to when a register or signal is being accessed outside of the scope of a particular thread's code. The following sections provide several specific examples as to when the user needs to intervene in live range analysis.

Asynchronous I/O operations

Asynchronous I/O operations are those that read or write into a variable not explicitly under compiler control. Such situations arise mostly commonly with the use of the “sig_done” token with memory intrinsics, but also in cases where a signal or transfer register is defined on one Microengine but accessed from another.

Example 1:

```
SIGNAL sig1;
SIGNAL_PAIR sig_pair;
__declspec(sram_read_reg) int sr1[4];
__declspec(sram_read_reg) int sr2[4];
sram_read(sr1, 0, 4, sig_done, &sig1);
dram_read_S(sr2, 0, 4, sig_done, &sig_pair);
wait_for_all(&sig1, &sig_pair);
sum += sr1[0] + sr2[0];
```

At first glance, nothing seems amiss, but in reality the compiler will determine that the user is not using the entirety of the buffers sr1 and sr2 and truncate the live range of individual members of an aggregate accordingly. That is, we write into these variables with the sram and dram reads, but do not consequently read all of the array elements (in this snippet, only sr1[0] and sr2[0] are read). So, it may attempt to conserve registers by assigning overlapping register ranges to sr1 and sr2. For example, sram transfer registers \$0 through \$3 may be assigned to sr1, and \$1 through \$4 may be assigned to sr2.

This is correct if the two memory reads complete in order, since the sum operation only uses sr1[0] and sr2[0], which are \$0 and \$1, respectively. But if the sram_read() and dram_read_S() operations complete out of order, then this assignment will cause problems. Specifically, when the dram_read_S() operation completes, the data the user needs will be read into \$1. But when the sram_read() operation completes, the contents of \$1 will be overwritten by the four-word read operation starting at \$0. In short, the compiler does not rely on characteristics of specific implementations of the network processor, but rather makes decisions based upon the general network processor architecture and the syntax of the C language.

The user must avoid this situation by informing the compiler that the entirety of both transfer buffers is being used with the __implicit_read() intrinsic.

```
wait_for_all(&sig1, &sig_pair);
sum += sr1[0] + sr2[0];
__implicit_read(&sr1);
__implicit_read(&sr2);
```

In this way, the compiler will extend the live range of all array elements of sr1 and sr2, and thereby not overlap the register allocation.

Of course, if the example above only included the one asynchronous `sram_read` and `wait_for_all` pair, there would be no problem. It is usually only in the presence of multiple asynchronous memory operations could there be a problem. But as a general rule of thumb, when using a read memory intrinsic with the `sig_done` token, a user should place an `__implicit_read` after the matching `wait_for_any` or `wait_for_all` to manually extend the live range of the memory variable to the correct point in the code.

Other asynchronous reads and writes to transfer registers or signal variables can occur under the following situations:

- A signal or transfer register that is defined on a remote ME and used on local ME - the definition/write is not visible from the local program
- A signal or transfer register is defined locally and used on a remote ME - the reference/read is not visible from the local program
- Special chip hardware that is designed to “push” data into a transfer register or send a signal, such as the receive state machine of the MSF

Most I/O instructions can overwrite the ME/CTX/XFER through the use of the `indirect_ref` token, causing the signal and/or transfer register to be used in the operation another Microengine or context – and therefore must be defined there. Additionally, the use of the reflector hardware implicitly will read or write from the transfer register of one thread to another. In the case of signal variables, asynchronous “reads” or “writes” can also be the result of using local or chip-wide thread signaling mechanisms.

Note: A “write” of a signal variable is defined to be the point at which a programmer asks for a signal to be generated from a chip resource, such as a memory controller. A “read” of a signal variable is defined to be the point at which a programmer waits on a signal to return (`ctx_arb`), or branches on a signal (`br_signal` or `br_!signal`).

Referring to a variable indirectly

Normally, all access to a variable declared to be in a register or signal storage class in Microengine C is done by explicitly referencing the variable name. For example, one can declare a variable in an sram transfer register and assign it a value with the following code:

```
__declspec(sram_write_reg) U32 wr_xfer0;  
wr_xfer0 = 0x1234;
```

But access to some types of variables can be done without referring to the variable’s name through special hardware support. Some of the software techniques used in conjunction with this support is used for performance reasons, while in some cases, it is actually required to perform a certain hardware function. However, these hardware/software constructs hold a special challenge for the compiler to correctly determine the live range of the variables in question.

The following cases fall into this category:

- A signal or transfer register is assigned an absolute register number and read/written without referring to the symbolic name
- A signal has the signal number exposed through signals() or signal_number() and read/written without referring to the symbolic name - for example, though local_csr_wr
- A xfer register has address taken and used in indexing reference through T_INDEX
- A NN register that being referenced indirectly though NN register ring
- A transfer register is read or written or a signal is sent via the cap calculated addressing instruction from another thread

Example 2:

```

__declspec(sram_write_reg) U32 wbuf[4];
__declspec(scratch) U32 sc_fun[16];
U32 ti0 = ((__ctx())<<4) | __xfer_reg_number(wbuf) << 2;
__asm {
    local_csr_wr[T_INDEX, ti0]
    nop
    nop
    nop
    alu[*$index++, --, B, sc_fun[index]]
    alu[*$index++, --, B, sc_fun[index+1]]
    alu[*$index++, --, B, sc_fun[index+2]]
    alu[*$index++, --, B, sc_fun[index+3]]
}
sram_write(wbuf, addr, 4, ctx_swap, &sig);

```

In this case, the compiler would make the determination that the live range of the wbuf array elements would start and end at the sram_write call. That is, it would only determine that wbuf was being read – as a result of the sram_write call, but not ever written into – as the write happens through the use of the indirect addressing capabilities of the hardware. Consequently, the compiler would only allocate a physical register to wbuf on the one line, missing the write into wbuf via the TINDEX CSR.

The correct course of action is to place an __implicit_write() intrinsic before the point of writing into wbuf indirectly:

```

__implicit_write(wbuf);
__asm {
    local_csr_wr[TINDEX, wbuf]
    nop
    // etc..

```

Now the live range of wbuf will start at the `__implicit_write` call and correct register allocation will take place.

Example 3:

```
SIGNAL sigRxEvent;
__declspec(sram_read_reg) rsw_pos_phy_t rsw[2];

int rxEventAddr    = __signal_number(&sigRxEvent);
int rsw0Addr       = __xfer_reg_number(&(rsw[0]));
rxFreeListReg      = rsw0Addr | (thread << THREAD_BITS) | \
(this_me << ME_NUM_BITS) | (rxEventAddr << SIGNAL_NUM_BITS);

// Add the thread to the receive freelist
AddThreadToFreelist(rxFreeListReg);

// Wait for signal from MSF receive hardware
wait_for_all(&sigRxEvent);
```

Another alternative to using the `__implicit_write()` and `__implicit_read()` intrinsics is use of the `volatile` keyword. The `volatile` keyword essentially extends the live range of the variable through the entirety of the program, bypassing live range issues altogether.

However, this method is not recommended in general for variables to be allocated to registers or local memory as these scarce resources will be eaten up very quickly this way! That said, for variables that truly need a live range over the whole program, declaring them to be `volatile` (as in the case of the `rsw` variable in the above example) is the alternative to placing a pair of `__implicit_write()` and `__implicit_read()` at the beginning and end of a program.

So in the example above, declaring the signal and transfer registers used with the receive hardware to be `volatile` assures correct code behavior.

```
volatile SIGNAL sig_RxEvent
volatile __declspec(sram_read_reg) rsw_pos_phy_t rsw[2];
```

Memory Intrinsics with Variable `ref_cnt`

The various memory intrinsics provide unfettered access to all of the special features provided by the memory controllers. For intrinsics that take a “count” parameter, it is preferred that a programmer uses a constant reference count in the arguments. If not, the compiler will generate a memory reference with an `indirect_ref` token, which may or may not lead to the desired results. The problem is that the compiler does not know the exact size of the transfer register buffer to be used in the operation and will make a conservative guess of one without programmer intervention.

Example 4:

```

void main()
{
    __declspec(sram_read_reg) mdata[4];
    unsigned int data_cnt;

    mdata[0] = 0x29;
    mdata[1] = 0x39;
    mdata[2] = 0x49;
    mdata[3] = 0x59;

    // Assume data_cnt has not been optimized to a constant
    sram_write(mdata, addr, data_cnt, ctx_swap, sig_srwr);

    // More code...
}

```

When the compiler calculates live range for the mdata array elements, it makes the conservative guess that only mdata[0] is being read with the sram_write intrinsic. Unless mdata[1], mdata[2] and mdata[3] are read later by another instruction, their live range will begin and end with their assignment, and unknown data will be written to sram.

There are two courses of action one can take. One is to use an __implicit_read to extend the live range of all members of the mdata array:

```

sram_write(mdata, addr, data_cnt, ctx_swap, sig_srwr);
__implicit_read(mdata);

```

The other option is to use the indirect reference form of the memory intrinsic:

```

srsw.refcount = data_cnt;
srsw_ind.ov_ref_count = 1;
sram_write_ind(mdata, addr, 4, srsw_ind, ctx_swap, sig_srwr);

```

The third parameter is the maximum number of longwords that could be written and must be a constant. The compiler will use that value as the potential length of data buffer used in the operation – even if at run-time less longwords are actually written to memory.

Finally, note that the -Qperinfo=128 command line option will warn a user if the compiler cannot determine the size of a memory option and needs to generate an indirect_ref token. An example of the output is below:

```

myfile.c(45): warning: sram_write(): Size of data access cannot be
determined at compile-time. __implicit_read/write may be needed to
protect xfer buffer. Use of sram_write_ind() is recommended instead.

```

Again, the aim of this warning is to remind the user to more precisely provide the livrange information for the variable used in the intrinsic. It is recommended that this option be used for all compilations.

Live Range Summary

The live range of a variable starts at the first point in the code it is written to and ends at the last read. In the case of signal variables, this should be interpreted as from when one asks for a signal to when that signal is consumed.

In special cases, the compiler cannot determine the live range of a variable correctly by itself; the most common being asynchronous I/O references and access to variables through indirect addressing. For these instances, the programmer needs to intervene with the use of either the `__implicit_read()` or `__implicit_write()` intrinsics or the `volatile` keyword to extend or truncate the live range of the variables in question.

Data Alignment

The question of how and where data is placed into various storage types is of utmost importance for two reasons. First is to make the most efficient use of a given, and presumably scarce, storage type. And the second is to guarantee the correct behavior of the compiler accessing variables indirectly (i.e. through pointers) or data not allocated by the compiler (i.e. packet data off the wire).

In short, misunderstanding how the compiler allocates variables to a storage type and with what alignment not only has performance implications, but could also affect the proper behavior of your program.

As such, one should first review the sections on alignment in the Microengine C Compiler Reference Manual. Then consider the examples in the remainder of this section:

Example 5:

```
typedef struct
{
    char  member1;
    int   member2;
} data1_t;

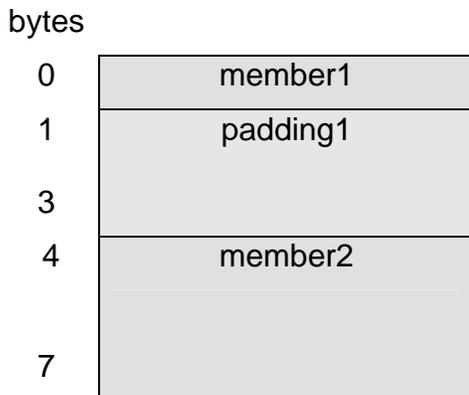
__declspec(scratch) data1_t good_data;
```

The natural alignment for the `data1_t` structure is on a four-byte boundary because the compiler will place the head of a structure aligned on a boundary relative to its storage

type. Large objects in DRAM (\geq sixteen bytes) will be aligned on a sixteen-byte boundary while all others, including anything declared in scratch, sram, local memory or any type of registers will be aligned on a four-byte boundary.

In addition, the compiler adds padding between elements to maintain individual member natural alignments. In this case, three bytes of padding will be inserted in between member1 and member2 so that member2 will fall on a four-byte boundary (since it is an "int").

The resulting good_data variable laid out in scratch memory like this:



For a programmer, there could be two issues here. One is the insertion of three bytes of padding between member1 and member2, and two is the potential performance penalty for accessing a structure on a boundary less than the address granularity of the storage type. Although in this example the structure is aligned well relative to its storage type (scratch), there are examples in the following sections to discuss this concern.

Packing Structures

Of course, the compiler adds the padding to improve structure access performance, but the most obvious issue with the compiler adding padding to a declared structure is that the size of the structure will be increased accordingly. This might not be a big deal when the structure is being allocated to a larger storage area like DRAM, but if this structure is destined for registers, it could lead to running out of a storage resource prematurely.

Another, less obvious issue with the compiler adding padding to a structure is that overlaying this structure type on top of non-compiler allocated data will result in incorrect access to members of that structure.

Example 6:

```
typedef struct
{
    U32 mac_addr[3];
    U16 prot_type;
    U32 src_addr;
    U32 dest_addr;
} hdr_t;

__declspec(dram) long long *p_packet;
__declspec(dram) hdr_t *p_header;

// Assign p_packet a packet buffer address.
// Then receive a packet into the RBUF.

dram_rbuf_read_ind(p_packet, ... );
p_header = (__declspec(dram) hdr_t *) p_packet;
if(p_header->dest_addr == OK_ADDRESS) { ... }
```

Consider that the compiler will add two bytes of padding between prot_type and src_addr to make sure that src_addr is aligned on a four-byte address. When mapped onto the packet data from the RBUF, there will be some misalignment of structure members.

A hdr_t data structure as the compiler is expecting it:

bytes	0 11	12 13	14 15	16 19	20 24
	mac_addr	prot_type	padding	src_addr	dest_addr

Actual header from a packet moved from the RBUF to DRAM (no padding!):

bytes	0 11	12 13	14 17	18 21
	mac_addr	prot_type	src_addr	dest_addr

From the above diagrams, it can be seen that accessing src_addr and dest_addr using the hdr_t pointer will return incorrect data. Specifically, p_header->src_addr will return the last two bytes of the actual src_addr and the first two of the actual dest_addr concatenated and p_header->dest_addr will return the last two bytes of the actual dest_addr and the next two bytes of the rest of the packet.

The compiler provides a means for the programmer to specify that no padding should be inserted between bit fields or between any members of a structure. Specifically the __declspec keywords of “packed” and “packed_bits” provide this functionality. In the example above, a programmer would do well to declare hdr_t as such:

```
typedef struct __declspec(packed)
{
    U32 mac_addr[3];
    U16 prot_type;
    U32 src_addr;
    U32 dest_addr;
} hdr_t;
```

Now the compiler will not insert any padding between `prot_type` and `scr_addr`, matching the “real”, non-compiler maintained data from the wire. Access to all members of the structure will be logically successful.

Of course, one of the drawbacks to packing data structures is that accessing members that cross an addressing boundary of the storage type (i.e. 4-byte for registers, SRAM and SCRATCH; 8-byte for DRAM) will incur extra overhead of bit extraction and/or concatenation and, possibly the reading/writing of extra memory locations. However, it is a necessary technique for the sorts of situations mentioned above.

Overriding Natural Alignment

Consider the application of a `__declspec(packed)` modifier to the structure of **Example 5**. Now, since padding between `member1` and `member2` will be removed, access to the second member of this structure will most likely lead to less than optimized code. The compiler will need to account for the fact that “`member2`” will span two 32-bit words. Still, access to the head of the structure (say, as part of an array) will be aligned to the storage type.

However, there are cases where access to both the structure members and the whole structure itself is not optimized. This is most likely to happen when dealing with packed structures full of small elements, as in the following example:

Example 7

```
typedef struct __declspec(packed)
{
    char  a_count;
    short s_count;
    char  b_count;
} count_t;

__declspec(sram1) count_t *pkt_count1;
__declspec(sram2) count_t *pkt_count2;
... // Assign values to pkt_count1;
*pkt_count2 = *pkt_count1;
```

Since the natural alignment of this structure is on a byte boundary, the compiler will make no assumptions about the position of the structure in memory. Consequently, it will generate code for the copy operation that will take into account the fact that the structure might span across two memory words. Structure alignment is not optimized automatically because of the possibility that the structure will be embedded inside an array or another structure, calling for the use of the structure's natural alignment.

However, a user can override the default alignment of a basic or aggregated data type using the “aligned(n)” `__declspec` modifier. If the unmodified structure's natural alignment is less than the addressable granularity of its storage region, the performance of whole structure copies can be improved by increasing the alignment to at least this granularity. That is, if the structure is being allocated to registers, SCRATCH or SRAM memories, performance would be improved if the structure was aligned on at least a four-byte boundary, and similarly on an eight-byte boundary for DRAM. To return to the original example, a better way to typedef the `count_t` structure would be:

```
typedef struct __declspec(packed aligned(4))
{
    char  a_count;
    short s_count;
    char  b_count;
} count_t;

*pkt_count2 = *pkt_count1; // copy performance is improved
```

More information on the syntax of the `aligned(n)` modified can be found in the Microengine C Compiler Support Reference Manual.

Efficient Structure Access

Structures make for convenient vessels to access related pieces of data. With gratuitous use of structures in most Microengine C programs, the compiler absolutely needs to be able to access to fields of a structure (including bit fields) efficiently. That said, there are a few things for a Microengine C programmer to keep in mind when designing and accessing data structures to help the compiler do so.

Sizing of Structure Members

Structure members with the following characteristics will produce the most efficient access because of the sizing of registers in the network processors:

- A multiple of four bytes in size
- Fall on a four byte offset from the start of a structure
- Do not cross a 4 byte boundary

For members lacking in one or more of these conditions, the compiler will need to generate extra instructions or memory accesses to extract and/or concatenate data from one or more registers every time this data is referenced. The following examples will demonstrate and discuss the repercussions for variables in this category.

Example 8:

<pre>typedef struct __declspec(packed) { char mem1; int mem2; int mem3; } mem_test_t;</pre>	<pre>typedef struct { int mem2; int mem3; char mem1; } mem_test_t;</pre>
---	--

The structure on the left does not follow the three guidelines above for any of its members. In fact, the members with the most overhead for access are mem2 and mem3, since both do not start on a four-byte offset and both cross a four-byte boundary (mem2 lies between byte offset 2 and 5, and mem3 lies between byte offset 6 and 9). The first member, mem1, is not a multiple of four bytes in size, but is an example of the second best structure construct. Members, including bit fields, of byte multiples in size (8, 16 or 24 bits) that follow the other two edicts only require a single `ld_field` instruction to operate on such data. And although not in this example, note that bit fields between 1 and 7 bits can be extracted with a single instruction with immediate mask. However an insert will take 2 instructions.

Of course, by default (i.e. without the `packed` keyword), the compiler would have placed 24 bits of padding between mem1 and mem2 in order to align mem2 and mem3 on their natural four-byte boundaries. This would have yielded optimal access to all members of those sizes in the structure at the cost of restrictions on how the programmer could use this structure with non-compiler maintained data (see [Section 2.2](#)). In either case, the total size of the structure would still be 12 bytes.

The data structure on the right allows the best possible access for structure members of those sizes, simply by reordering their declared positions. Here, all members are aligned on byte boundaries and are of proper sizing to warrant optimal treatment by the compiler, similar to a non-packed version of the first structure – but without the padding between members.

Of course, a programmer can create structures with any member sizing and order and the compiler will do its best to optimize access, but the best possible performance will be obtained with a little forethought upfront. For some, total structure size or pad-less data will be a concern and so packing structure members will be necessary at the cost of a few extra instructions to pack and unpack data. But if at all possible, a software architect should adhere to the three guidelines concerning structure layout presented above.

The Use of Unions

As an alternate, or perhaps a supplement to the structure member sizing strategies outlined in the previous section, unions can be used to streamline access to several structure members at once. This can be critically important if your structure resides in a high latency memory.

Example 9:

```
typedef struct {
    union
    {
        struct {
            U32 a1:16;
            U32 a2:16;
            U32 a3:16;
            U32 a4:16;
        } a_params;
        struct {
            U32 b1;
            U32 b2;
        } b_params;
    };
} params_t;

volatile __declspec(sram) params_t param_set;

void main()
{
    volatile U32 p1 = 0x1111;
    volatile U32 p2 = 0x2222;
    volatile U32 p3 = 0x3333;
    volatile U32 p4 = 0x4444;

    // Assign each member separately
    param_set.a_params.a1 = p1;
    param_set.a_params.a2 = p2;
    param_set.a_params.a3 = p3;
    param_set.a_params.a4 = p4;

    // Format data for a1 and a2 and write into b1
    param_set.b_params.b1 = (p1<<16) | p2;

    // Format data for a3 and a4 and write into b2
    param_set.b_params.b2 = (p3<<16) | p4;
}
```

Although the compiler may be able to figure out how to combine the “a_params” code above, it might generate four separate writes to sram, one for each of the a_params’ assignment statements:

```

/*****/          param_set.a_params.a1 = p1;
alu_shf[$0, --, B, a6, <<16]
sram[write, $0, a3, 0, 1], ctx_swap[s2]

/*****/          param_set.a_params.a2 = p2;
alu[$0, --, B, b7]
sram[write, $0, a3, 0, 1], ctx_swap[s2]

/*****/          param_set.a_params.a3 = p3;
alu_shf[$0, --, B, a7, <<16]
sram[write, $0, a3, 4, 1], ctx_swap[s2]

/*****/          param_set.a_params.a4 = p4;
alu[$0, --, B, b0]
sram[write, $0, a3, 4, 1], ctx_swap[s2]

```

A less straightforward example would include other code between each assignment, making it more difficult for the compiler to even consider any memory access optimizations.

In any case, a reduction in memory accesses is guaranteed through the use of the union in the params_t structure. Here, the four 16-bit a_params members are unioned with two 32-bit b_params members. By using the later as an alias to write into the former, four sram writes become two. Of course, the programmer will need to format the data “by-hand”, explicitly performing the shifting and logical bit operations in code versus letting the compiler generate such code automatically. But the tradeoff in code complexity to save accesses to memory is well worth it – just be sure to comment appropriately to avoid confusion.

```

/*****/          param_set.b_params.b1 = (p1<<16) | p2;
alu_shf[$0, b7, OR, a6, <<16]
sram[write, $0, a3, 0, 1], ctx_swap[s1]

/*****/          param_set.b_params.b2 = (p3<<16) | p4;
alu_shf[$0, b0, OR, a7, <<16]
sram[write, $0, a3, 4, 1], ctx_swap[s1]

```

Miscellaneous Considerations

These tips did not fall under one of the main headings, but are helpful to maximize code performance.

Signed vs. Unsigned Integers

The various basic datatypes, such as integers (“int”) are, by default, signed entities. In some cases, a small performance benefit can be derived by using the unsigned versions wherever possible, especially in bitfield structs. Otherwise the compiler may generate unnecessary ASR instructions to handle sign extension when extracting fields.

Example 10:

```
typedef struct three_fields
{
    int a1:16;
    int b2: 8;
    int c3: 8;
} three_fields_t;

int result;
three_fields_t my_var;

result = my_var.b2; // implemented with 2 instr, ASR, ALU_SHF
```

Instead, the preferred method would be to declare three_fields_t with unsigned integer bit fields as such:

```
typedef struct three_fields
{
    unsigned int a1:16;
    unsigned int b2: 8;
    unsigned int c3: 8;
} three_fields_t;
```

And so the following access to a field of such a structure will only take 1 instruction, instead of two:

```
result = my_var.b2; // implemented with 1 instr, LD_FIELD
```

Tuning Established Code

After following the good programming practices expounded by the Microengine C Compiler Support Reference Manual and this document, a programmer should be able to program logically sound code for the network processors. This next section aims to bring the programmer to the next level in terms of coding conventions that will lead to best compiler, and hence, best program performance. It will highlight techniques to handle register spillage, to improve access to variables in memory, to optimize function calls and more.

Register Spillage

Not all register candidate variables will be allocated to actual registers by the compiler either because there simply are not enough registers to handle all variables live at a certain point in the code or because the address of a variable was taken. In such a case, the default behavior of the compiler is to automatically “spill” these variables to one of the following resources based upon the `-Qspill=n` command line option:

- Next Neighbor registers
- Local Memory
- SRAM Memory

Please see the compiler reference manual for more information on when the compiler will spill variables, as well as the `-Qspill` command line option.

When register spillage occurs, the compiler will provide the information of which variables were spilled, and to which storage type if the `-Qperinfo=1` command line option is used. In addition, the `-Qliveinfo` option provides liveness information for all register variables in the program. More information on these options can be found in the compiler reference manual and in following sections of this document.

Handling Register Spillage

One option for the programmer is to turn off the automatic spillage feature of the compiler altogether using `-Qspill`. In this case, if the compiler cannot allocate all variables without explicit storage declarations to registers, the compilation will fail and the programmer will have to perform the rearrangement of data by hand. This is a good option if you want the programmer to have absolute control over the storage regions for all variables at all times. However, there are less severe options that can give a variable-by-variable or code section by code section level of control for the compiler's opportunities for register spillage.

First, when a programmer absolutely needs an individual variable to be placed in a register at all times in the program (ex. frequently accessed variables), the programmer can explicitly declare the variable with a `gp_reg` storage type.

Example 11:

```
unsigned int count1;           // a register candidate
__declspec(gp_reg) unsigned int count2; // must go to a GPR!
```

If for some reason the compiler cannot allocate the count2 variable to a register, then the compilation will fail. The programmer will then need to rearrange the code to use less register variables during the live range of the failed variable.

There could be specific sections of code for which the programmer does not want to see any variables spilled. The `__no_spill_begin()` and `__no_spill_end()` intrinsic functions provide this functionality. In this way, the no-spilling directive to the compiler is done relative to a section of code versus on a per variable basis. For example, a variable with several live ranges could spill in one section of code, but not in any `__no_spill` regions. Again, if the compiler cannot figure out how not to spill variables in a `__no_spill` region, the compilation will fail.

Note: The current implementation of the compiler does not spill variables accessed inside a `no_spill` region for the entirety of the program (i.e. not just inside the `no_spill` region). This is not optimal behavior and will be addressed in future releases of the compiler.

Functions

Due to the lack of a stack in the network processors, the compiler has to incur some overhead for function calls. And although the compiler has many optimizations to affect run time performance as little as possible, several programming techniques will provide the compiler with the most opportunity to do so.

Inlining Functions

The function calling convention in Microengine C is to pass as many enregisterable arguments as possible, saving the return PC to a register, then performing a hard branch to the function. This can be quite a bit of overhead, especially in relation to functions with few lines of actual code.

The alternative is to have a function call inlined at the place in the code in which it was called. In this case, the compiler does not waste a register unnecessarily for saving the PC, or waste execution time branching to the function and back.

The inlining of functions is controllable through compiler options as well as through the use of directives in the C source code. The `__forceinline` keyword forces the compiler to inline the function regardless of the size of the function as long as inlining has not been turned off via the `-Obn` compiler switches or in debug code via the `-Od` switch. The `__inline` keyword allows the compiler to decide whether or not to inline the function based on cost/benefit analysis performed by the compiler when explicit inlining is enabled (`-Ob1`).

On the other hand, a programmer can prevent the compiler from inlining a particular function while still enabling the general inlining capabilities of the compiler through the use of the `__noinline` keyword preceding the function prototype and definition. Although the compiler tries to balance control store versus performance based upon command line compiler options, the use of this keyword allows a programmer to precisely control inlining on a function by function basis.

But in general, use the `-Ob2` command line switch to allow the compiler to inline shorter functions automatically based upon compile time heuristics.

Note: The current implementation of the compiler will still inline functions defined with the `__forceinline` keyword even with the `-Od` switch specified. This behavior is so to support backward compatibility for older versions of software, but is subject to future changes.

Optimizing Pointer Arguments

It is sometimes possible to improve the speed of access to function arguments passed in with pointers.

Example 12:

```
void foo(MyStruct *p_x)
{
    // some code using *p_x and assigning *p_x
}
void main()
{
    myStruct_t x;
    ...
    foo(&x);
    ...
}
```

In this example, the user wishes to use the function “foo” to modify the contents of the structure “x”, by passing the address of x to foo. Since general-purpose registers cannot be accessed with pointers, the compiler cannot place the structure x into registers. Rather, x will be allocated into other storage regions such as local memory or sram, slowing down – potentially significantly - access to the data contained in x.

If the programmer can guarantee that the pointer parameter of the function is not accessed through “unknown” means (for example through another pointer whose definition is ambiguous or from another thread), then the “restrict” qualifier is placed directly before the parameter in the function definition. In doing so, the compiler will automatically perform a “structure copy optimization” which will copy the structure to be passed to a global temporary structure accessible by the function foo. Both the original

and temporary structures can be placed into registers, with a significant performance gain over a non-restricted pointer parameter. In this case, the function definition would look like this:

```
void foo(MyStruct * restrict p_x)
{
    // Alias-free code using *p_x and assigning *p_x
}
```

Again, the restrict keyword should only be used if the programmer can guarantee controlled pointer access to the data structure in question. The compiler reference manual provides a list of allowable operations for a restricted pointer, and a command line switch “-Qperfinfo=256” to help determining any violations of these rules. But remember that ultimately the programmer is responsible for the safe application of the restrict keyword.

Miscellaneous Optimizations

These tips did not fall under one of the main headings, but are helpful to maximize code performance.

Conditional Statements

Compare to zero (==, !=, <, >) rather than the explicit value when possible. This allows the condition codes to be tested as opposed to the compiler generating a subtract and then testing of the condition codes.

Example 13:

```
if( queue_entry->current_buf.sop_flag == 1 )
{
    do_something();
}
```

In this case, the compiler will need to generate an extra alu instruction to perform the subtract-compare. A better implementation is as follows, and uses one less instruction:

```
if( queue_entry->current_buf.sop_flag != 0 )
```

Compiler Defer Slot Filling

In general, the algorithm used in the compiler to fill defer slots is limited to looking in the basic block immediately above or below a branch or context swap.

Example 14

```
void cool_function(U32 pass)
{
    if(pass)
    {
        ...
    }
    else
    {
        ...
    }

    gl_foo = gl_a + gl_b;
    gl_bar = gl_x + gl_y;
}
```

Presumably, the if statement will be translated into a branch if equal (BEQ) instruction, which allows for up to three instructions in the branch shadow to be deferred. However, the code above does not give the compiler any instructions as candidates to place into the branch defer slots. This, of course, supposes that the code in the if and else blocks cannot be moved or are otherwise not candidates to lie in the branch shadow. But if the gl_foo and gl_bar assignments can be moved freely in the code from a logical standpoint, then placing them directly above the if statement will provide the compiler with two more opportunities for optimization.

```
gl_foo = gl_a + gl_b;
gl_bar = gl_x + gl_y;
if(pass)
{
// etc...
```

Debugging Techniques

Compile-time Information

The Microengine C compiler can produce debugging information into the .list file output to be passed to the linker, including source file to assembly code mapping. In addition, the compiler will produce a separate .dbg file, used by the Developer's Workbench in conjunction with the .list files to provide source-level debugging capabilities. Specifically, the .dbg file contains variable scope information and datatype definition (ex. structure field layout) for use in the Data Watch window. In addition, an optional command line switch prints out various information to help the programmer make optimal decisions on topics ranging from register spillage to "restrict" pointer violations.

Performance Information

The `-Qperfinfo=n` command line switch can provide one or more of the following pieces of compile time information:

- n=0 - no information (similar to not specifying)
- n=1 - register candidates spilled and where to
- n=2 - instruction-level symbol liveness and register allocation (**obsolete!**)
- n=4 - function-level symbol liveness and register allocation (**obsolete!**)
- n=8 - function sizes
- n=16 - local memory allocation
- n=32 - live range conflicts causing SRAM spills
- n=64 - instruction scheduling statistics
- n=128 - Warn if the compiler cannot determine a memory I/O transfer size
- n=256 - Display information for "restrict" pointer violations
- n=512 - Print offsets of potential jump[] targets
- n=2048 - Print maximum physical register pressure

Notice each n value above is actually a bit mask for the `-Qperfinfo` switch. That is, a user can request multiple informational items from the above list for a given compilation by OR'ing several n values together. For example, if a user would like to view the register candidates spilled, local memory allocation and warnings for "restrict" pointer violations during a compilation, add `-Qperfinfo=273` to the compiler command line.

It is recommended that all compilations include n=1 if register spillage is enabled (via the `-Qspill` switch) as spilled variables will have various performance implications. Similarly, other options should be included if the code warrants it (i.e. use n=256 if there are restricted pointer parameters in your program).

These first three options, along with the last one, had been provided to help the programmer manage register allocation in the program. However, `-Qperfinfo=2` and `-Qperfinfo=4` have been superceded by a new command line switch, `-Qliveinfo=[gr,sr,sw,srw,dr,dw,nn,sig,all]` to print out liveness information for all register allocated variables in a more helpful and user-friendly manner. In fact, it uses a

different algorithm – one that more accurately reflects real register allocation – than –Qperfinfo=2 or –Qperfinfo=4 did. A user can display the register allocation information for only the register types of interest, by providing one or more of the following –Qliveinfo options:

```
gr: GPRs
sr: SRAM read regs
sw: SRAM write regs
srw: SRAM read/write regs
dr: DRAM read regs
dw: DRAM write regs
drw: DRAM read/write regs
nn: NN registers (self mode only)
sig: signals
all: all registers
```

In short, the –Qliveinfo switch can help the user analyze their program and determine which code segments have a high “register pressure” and need to be restructured.

The first section of compiler output from the –Qliveinfo=gr command line switch details, on a function by function basis, the registers live when the function is called (“Live in”), those live upon completion of the function (“Live out”) and those live both in and out of the function (“Live through”).

Example 15:

: Live info. of gpr registers for Function meter_calculate_ebs_cbs:

```
: Live in(11):
:   gr.554(_timestamp) gr.555(_timestamp+4)
gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr)
gr.645(entry) gr.685(result) gr.687(p_sram) gr.740(cgt.1090)
gr.743(cgt.1093) gr.897(..) gr.899(..)
: Live out(9):
:   gr.554(_timestamp) gr.555(_timestamp+4)
gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093)
gr.899(..)
: Live through(7):
:   gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093)
gr.899(..)
```

The registers are printed out in the following format:

```
cls.ID(variable_name)
```

where cls is one of the register classes mentioned above, the ID is a compiler maintained “virtual register” number and variable_name is the name of the

corresponding variable. A variable_name of “.” implies a compiler generated temporary variable is being used.

Use this information to help determine which functions have maxed out, or are close to maxing out, register usage. For example, there are 32 available GPRs per thread, so seeing values close to 32 in the Live parentheses above advises a programmer to pay special attention to those functions if there is a problem with register spillage.

Following the first section is register liveness on a per-instruction basis. For each line of code in the program (Microengine C source with corresponding assembly), the “Live set” of registers is listed. This information lets a programmer further refine the search for high register pressure areas of code.

Example 16:

```

:  /*****/  meter_params[entry].timestamp = timestamp;
:    alu[gr.955(..) , 4, +, gr.802(..) ]
:    Live set(15): gr.554(_timestamp) gr.555(_timestamp+4)
gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr) gr.651(tmp.27)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093) gr.802(..) gr.803(..)
gr.804(..) gr.897(..) gr.899(..) gr.955(..)
:
:    alu[??, --, B, gr.554(_timestamp) ]
:    Live set(14): gr.554(_timestamp) gr.555(_timestamp+4)
gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr) gr.651(tmp.27)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093) gr.803(..) gr.804(..)
gr.897(..) gr.899(..) gr.955(..)
: // etc...

```

Additionally, the `-Qperfinfo=2048` switch will provide a quick summary of the lines of code with the maximum physical register pressure. A example dump looks like this:

```

Maximum live physical gprs (52) at myfile.c 245
Maximum live physical gprs (52) at myfile.c 203
Maximum live physical gprs (52) at myfile.c 196
Maximum live physical gprs (52) at myfile.c 192

```

The maximum number of live physical GPRs before spilling is printed along with the file name and line number of the high register pressure points.

Run-Time Debug Tools

When Microengine C based programs are compiled to one or more Microengines with debug information turned on, the Developer's Workbench provides several debugging tools specifically for the compiler, allowing the programmer to debug on a source code or assembly code level. The Development Tools User's Guide describes the use of all such tools, including source/assembly view toggling, data watches and breakpoints.

Summary

A Microengine C programmer must be well versed in both the operation of the network processor hardware and Microengine C compiler options to write effective and performance minded code. Experience (i.e. writing code!) is the only way to become a great programmer, but this document is aimed at shortening the curve by providing explanations and examples for many of the common pitfalls and performance considerations for a new Microengine C user.

In addition, the explanations in plain language of the slew of information provided by the compiler will help a programmer make optimal decisions for data placement and code structure. And a discussion of the compiler specific debugging features and tools in the Developer's Workbench will assist a programmer to debug code quickly from both a logic and performance point of view.

The tools and documentation are there to support programmers to make the process of writing and debugging code as easy as possible, but ultimately, it takes a knowledgeable and well-disciplined programmer to write effective Microengine C code.