



Intel[®] IXP2400/IXP2800 Network Processors

Microengine C Language Support Reference Manual

November 2003



Revision History

Date	Revision	Description
January 2002	001	Pre-Release 2 (PR2)
May 2002	002	Release for IXA SDK 3.0
August 2002	003	Release for IXA SDK 3.0 Pre-Release 4
November 2002	004	Release for IXA SDK 3.0 Pre-Release 5
January 2003	005	Release for IXA SDK 3.0 Pre-Release 6
May 2003	006	Release for IXA SDK 3.1 for VmWorks
June 2003	007	Release for IXA SDK 3.1 Pre-Release 2
September 2003	008	Release for IXA SDK 3.5
November 2003	009	Release for IXA SDK 3.5

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.

The information in this manual 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.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The software described in this User's Guide may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

Intel SpeedStep, Intel Thread Checker, Celeron, Dialogic, i386, i486, iCOMP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Xeon, Intel XScale, Itanium, MMX, MMX logo, Pentium, Pentium II Xeon, Pentium III Xeon, Pentium M, and VTune are trademarks or registered trademarks 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.

Copyright © Intel Corporation 2002–2003.

Contents

1	Introduction	11
1.1	Purpose	11
1.2	Features.....	11
1.3	Nonfeatures	11
1.4	Conventions Used in this Manual	12
1.4.1	Version-Specific References	12
2	Overview	13
2.1	Network Processor Architecture Overview	13
2.1.1	Data Terminology	13
2.1.2	Register Model.....	13
2.1.3	Next Neighbor Registers.....	14
2.1.4	Local Memory	16
2.1.5	External Memory.....	16
2.1.6	FIFO Queues	16
2.1.7	Signals	17
2.1.8	Reflector	17
2.1.9	Indirect Register Access	17
2.1.10	Threading Model.....	17
2.1.11	Features Not Supported	18
2.2	Compilation Model	18
2.2.1	Number of Contexts	18
2.2.2	Inlining	19
2.3	Running the C Compiler	19
2.3.1	The Command Line	19
2.3.2	Supported Compilations	20
2.3.3	Supported Compiler Option Switches	20
2.3.3.1	Environment Variables	24
2.3.4	Input and Output File Types	25
2.3.5	Linking a Microengine .UOF file.....	25
2.3.6	Util.c.....	26
2.3.6.1	Utility Functions (util.c).....	26
2.3.6.2	Multi-threading restrictions.....	27
2.3.7	Example—Using the C Compiler.....	27
2.3.7.1	The C File	27
2.3.7.2	Compiling the File	28
2.3.7.3	Linking the File.....	28
2.3.7.4	Running the File.....	28
2.3.7.5	Initialization File	28
2.3.8	C Compiler Graphical User Interface from Developer Workbench	29
2.3.8.1	Build Features.....	29
2.3.8.2	Debug Feature	29
2.4	Running and Debugging Under the Developer Workbench.....	30
3	C Language Support	33
3.1	Standard Data Types	33
3.1.1	Basic Data Types.....	33

3.1.2	Pointer Representation	33
3.1.3	Bitfields	33
3.1.4	Floating Point Types	34
3.1.5	String Literals	34
3.1.6	Size of Data Types	34
3.1.7	Alignment of Data Types	35
3.1.8	Packed Aggregates	36
3.1.9	Pointer Alignment Assumptions and Unaligned Pointers	37
3.1.10	Endian Support	38
3.1.10.1	Compiler Limitations of Endian Support	38
3.2	Data Allocation	40
3.2.1	Register Regions	41
3.2.1.1	General Purpose Registers	41
3.2.1.2	Transfer Registers	41
3.2.1.3	Next Neighbor Registers	42
3.2.1.4	Volatile Registers	43
3.2.2	Memory Regions	43
3.2.3	Shared Data	44
3.2.4	Global data	45
3.2.5	Load Time Constants	45
3.2.6	Signals	45
3.2.6.1	Signal Variable Restrictions	46
3.2.7	Local Memory Allocation	47
3.2.7.1	Overview	47
3.2.7.2	Placement of Variables	47
3.2.7.3	Thread Local vs. Shared Storage	48
3.2.7.4	Viewing Local Memory Usage	48
3.2.7.5	Alignment Information for Local Memory Pointers	50
3.2.7.6	Suggestions for Improving Local Memory Use	50
3.3	Reflector Inputs/Outputs	51
3.4	Summary of Allowed Data Attribute Combinations	52
3.5	Expressions	52
3.6	Statements	52
3.7	Functions	52
3.7.1	Supported	52
3.7.2	Not Supported	52
3.7.3	Extended Function Attributes	53
3.7.4	Optimizing Pointer Arguments	53
3.7.4.1	The “restrict” Qualifier	54
3.8	User Assisted Live Range Analysis	55
3.9	Viewing Live Ranges	57
3.9.1	Limitations and Restrictions on Viewing Live Ranges	58
3.10	Critical Path Annotation and Code Layout	59
3.10.1	Multiple Critical Paths	61
3.11	User-Guided switch() Statement Optimization	62
3.11.1	Default Case Removal	63
3.11.2	Switch Block Packing	63
3.12	Creating Context Swap-Free Regions of Code	64
3.13	Loop unrolling control	64
3.14	Mixing C and Microcode in One Microengine	66
3.14.1	Command Line Options and Usage model	66

3.14.2	Naming and Calling Conventions	67
3.14.2.1	Register Variable Naming Conventions	67
3.14.2.2	Sharing Variables Between C and Assembly	68
3.14.2.3	Calling Conventions	68
3.14.3	Mixed C and Microcode Examples	69
3.14.3.1	Function Parameter Passing	69
3.14.3.2	Register Usage	70
3.14.4	Restrictions on Mixing C and Microcode.....	70
3.15	Unsupported ANSI C99 Features	71
4	Intrinsic Functions	73
4.1	Intrinsic Syntax Conventions	74
4.2	Unaligned Data Access	75
4.2.1	Unaligned Get Functions	75
4.2.2	Unaligned Set Functions.....	78
4.2.3	Unaligned Memory Copy Functions.....	80
4.3	Memory I/O Functions	82
4.3.1	Transfer Register Modifiers.....	82
4.3.2	Memory I/O Data types	83
4.3.2.1	sync_t.....	85
4.3.2.2	bytes_specifier_t.....	86
4.3.2.3	reflect_signal_t.....	87
4.3.2.4	pci_read_write_ind_t, sram_read_write_ind_t	88
4.3.2.5	scratch_read_write_ind_t, scratch_ring_ind_t, sram_read_qdesc_ind_t, sram_ring_ind_t, sram_journal_ind_t, cap_read_write_ind_t, msf_read_write_ind_t, reflect_read_write_ind_t89	
4.3.2.6	dram_read_write_ind_t	91
4.3.2.7	sram_atomic_ind_t.....	92
4.3.2.8	scratch_atomic_ind_t, sram_csr_read_write_ind_t, cap_csr_read_write_ind_t93	
4.3.2.9	dram_rbuf_tbuf_ind_t.....	94
4.3.2.10	sram_enqueue_ind_t	95
4.3.2.11	sram_dequeue_ind_t	96
4.3.2.12	hash_ind_t	97
4.3.2.13	generic_ind_t	98
4.3.3	Memory I/O Functions	99
4.3.3.1	Scratch Operations	100
4.3.3.2	SRAM Operations	138
4.3.3.3	DRAM Operations.....	204
4.3.3.4	MSF Operations.....	211
4.3.3.5	PCI Operations	222
4.3.3.6	Reflector Operations	227
4.3.4	Limitations on Some I/O Functions.....	232
4.4	Synchronization Functions.....	234
4.4.1	Synchronization Data Types.....	234
4.4.1.1	signal_t.....	234
4.4.1.2	SIGNAL_MASK.....	234
4.4.1.3	inp_state_t	234
4.4.2	Synchronization Functions.....	235
4.4.2.1	__signals()	236
4.4.2.2	signal_test()	237
4.4.2.3	ctx_swap().....	238
4.4.2.4	ctx_wait().....	239

4.4.2.5	__wait_for_any(), __wait_for_all().....	240
4.4.2.6	signal_same_ME	241
4.4.2.7	signal_same_ME_next_ctx	242
4.4.2.8	signal_prev_ME	243
4.4.2.9	signal_prev_ME_this_ctx	244
4.4.2.10	signal_next_ME	245
4.4.2.11	signal_next_ME_this_ctx	246
4.5	Control and Status Register (CSR) Access Functions	247
4.5.1	CAP Data Types	247
4.5.1.1	cap_csr_t	247
4.5.1.2	local_csr_t	248
4.5.1.3	cap_read_write_ind_t	249
4.5.2	CAP Functions	250
4.5.2.1	cap_csr_read(), cap_csr_read_D()	252
4.5.2.2	cap_csr_read_ind(), cap_csr_read_D_ind()	253
4.5.2.3	cap_csr_write(), cap_csr_write_D()	254
4.5.2.4	cap_csr_write_ind(), cap_csr_write_D_ind()	255
4.5.2.5	cap_read(), cap_read_D()	256
4.5.2.6	cap_read_ind(), cap_read_D_ind()	257
4.5.2.7	cap_write(), cap_write_D()	258
4.5.2.8	cap_write_ind(), cap_write_D_ind()	259
4.5.2.9	cap_fast_write()	260
4.5.2.10	local_csr_read()	261
4.5.2.11	local_csr_write()	262
4.6	Hash Access Functions	263
4.6.1	Data Types	263
4.6.2	Functions	264
4.6.2.1	hash_48(), hash_48_D()	265
4.6.2.2	hash_48_ind(), hash_48_D_ind()	266
4.6.2.3	hash_64(), hash_64_D()	267
4.6.2.4	hash_64_ind(), hash_64_D_ind()	268
4.6.2.5	hash_128(), hash_128_D()	269
4.6.2.6	hash_128_ind(), hash_128_D_ind()	270
4.6.3	Limitations on Hash Functions	271
4.7	CAM (Content Addressable Memory) Access Functions	272
4.7.1	Data Types	272
4.7.1.1	cam_lookup_t	272
4.7.2	Functions	273
4.7.2.1	cam_clear()	274
4.7.2.2	cam_lookup()	275
4.7.2.3	cam_read_tag()	276
4.7.2.4	cam_read_state()	277
4.7.2.5	cam_write_state()	278
4.7.2.6	cam_write()	279
4.8	CRC Access Functions	280
4.8.1	Data Types	280
4.8.1.1	bytes_specifier_t	280
4.8.2	Functions	280
4.8.2.1	crc_5_be()	284
4.8.2.2	crc_5_be_bit_swap()	285
4.8.2.3	crc_5_le()	286
4.8.2.4	crc_5_le_bit_swap()	287
4.8.2.5	crc_10_be()	288

4.8.2.6	crc_10_be_bit_swap()	289
4.8.2.7	crc_10_le()	290
4.8.2.8	crc_10_le_bit_swap()	291
4.8.2.9	crc_16_be()	292
4.8.2.10	crc_16_be_bit_swap()	293
4.8.2.11	crc_16_le()	294
4.8.2.12	crc_16_le_bit_swap()	295
4.8.2.13	crc_ccitt_be	296
4.8.2.14	crc_ccitt_be_bit_swap	297
4.8.2.15	crc_ccitt_le	298
4.8.2.16	crc_ccitt_le_bit_swap	299
4.8.2.17	crc_32_be()	300
4.8.2.18	crc_32_be_bit_swap()	301
4.8.2.19	crc_32_le()	302
4.8.2.20	crc_32_le_bit_swap()	303
4.8.2.21	crc_iscsi_be()	304
4.8.2.22	crc_iscsi_be_bit_swap()	305
4.8.2.23	crc_iscsi_le()	306
4.8.2.24	crc_iscsi_le_bit_swap()	307
4.8.2.25	crc_read()	308
4.8.2.26	crc_write()	309
4.9	Miscellaneous Functions	310
4.9.1	Functions	310
4.9.1.1	dbl_shr()	313
4.9.1.2	dbl_shl()	314
4.9.1.3	__ctx()	315
4.9.1.4	__ME()	316
4.9.1.5	__n_ctx()	317
4.9.1.6	__nctx_mode()	318
4.9.1.7	sleep()	319
4.9.1.8	ffs()	320
4.9.1.9	__LoadTimeConstant()	321
4.9.1.10	__global_label()	322
4.9.1.11	multiply_24x8()	323
4.9.1.12	multiply_16x16()	324
4.9.1.13	multiply_32x32_lo()	325
4.9.1.14	multiply_32x32_hi()	326
4.9.1.15	multiply_32x32()	327
4.9.1.16	__set_timestamp()	328
4.9.1.17	__timestamp_start(), __timestamp_stop	329
4.9.1.18	__set_profile_count()	330
4.9.1.19	__profile_count_start(), __profile_count_stop	331
4.9.1.20	__signal_number()	332
4.9.1.21	__xfer_reg_number()	333
4.9.1.22	__assign_relative_register()	334
4.9.1.23	__implicit_read()	335
4.9.1.24	__implicit_write()	336
4.9.1.25	__free_write_buffer()	337
4.9.1.26	inp_state_test()	338
4.9.1.27	bit_test()	339
4.9.1.28	nn_ring_dequeue_incr()	340
4.9.1.29	nn_ring_dequeue()	341
4.9.1.30	nn_ring_enqueue_incr()	342
4.9.1.31	byte_align_block_le()	343

4.9.1.32	byte_align_block_be()	344
4.9.1.33	__no_spill_begin()	345
4.9.1.34	__no_spill_end()	346
4.9.1.35	assert()	347
4.9.1.36	__critical_path()	348
4.9.1.37	pop_count()	349
4.9.1.38	__switch_pack	350
4.9.1.39	__impossible_path	351
4.9.1.40	__no_swap_begin	352
4.9.1.41	__no_swap_end	353
4.10	Restrictions On Intrinsic	354
4.10.1	Intrinsic Function Arguments that Map to Transfer Registers in Microcode	354
5	Inline Assembly Language	357
5.1	Single __asm Instruction	358
5.2	Block of __asm Assembly Code	359
5.3	Instruction Format	360
5.4	Operand Syntax	361
5.4.1	Register Operands	361
5.4.2	Immediate Operands	362
5.4.3	Usage Examples	362
5.5	Restrictions on Use Of Assembly Language	365
6	Compiler Optimizations	367
6.1	Machine Independent Optimizations	367
6.2	Network Processor Specific Optimizations	367
6.2.1	Registrations	367
6.2.2	Read/Write Combining	367
6.2.3	Peephole Optimization	368
6.2.4	Defer Slot Filling	368
6.2.5	Local Memory Grouping	368
6.2.6	Local Memory Autoincrement/Autodecrement Conversion	369
6.2.7	Scheduling	370
6.2.8	I/O Parallelization	370
7	Tips for Optimization, Troubleshooting, and Debugging	373
7.1	Optimizing Your Code	373
7.2	Things to Remember When Writing Microengine C Code	374
7.3	Troubleshooting	378
7.3.1	Program Does Not Fit	378
7.3.2	Program Does Not Run Correctly	378
7.4	Debugging Inline Functions	378
8	Mutual Exclusion Library	379
8.1	Introduction	379
8.2	MUTEXLV Usage	379
8.3	MUTEXG Usage	380
8.4	Functions	380
8.4.1	MUTEXLV_init (MUTEXLV)	380
8.4.2	MUTEXLV_destroy(MUTEXLV, MUTEXID)	380
8.4.3	MUTEXLV_lock(MUTEXLV, MUTEXID)	381

8.4.4	MUTEXLV_unlock(MUTEXLV, MUTEXID).....	381
8.4.5	MUTEXLV_trylock(MUTEXLV, MUTEXID, ERRCODE).....	381
8.4.6	MUTEXLV_testlock(MUTEXLV, MUTEXID, ERRCODE).....	381
8.4.7	MUTEXG_init (MUTEXG).....	382
8.4.8	MUTEXG_destroy (MUTEXG).....	382
8.4.9	MUTEXG_lock (MUTEXG).....	382
8.4.10	MUTEXG_unlock (MUTEXG).....	383
8.4.11	MUTEXG_trylock (MUTEXG, ERRCODE).....	383
8.4.12	MUTEXG_testlock (MUTEXG, ERRCODE).....	383
9	Semaphore Library	385
9.1	Semaphore Data Types.....	385
9.2	Semaphore Functions.....	385
9.2.1	SEML_init(SEML, SEMVALUE).....	385
9.2.2	SEML_destroy(SEML).....	385
9.2.3	SEML_post(SEML) SEML_dec(SEML).....	386
9.2.4	SEML_wait(SEML).....	386
9.2.5	SEML_trywait(SEML, ERRCODE).....	386
9.2.6	SEML_barrier(SEML,n).....	387
9.2.7	SEML_trybarrier(SEML, ERRCODE).....	387
9.2.8	SEML_getvalue(SEML).....	387
9.2.9	SEML_set_barrier_at(SEML,n) SEML_clr_barrier_at(SEML,n).....	387
A	-Qperfinfo Output Information	397
A.1	-Qperfinfo=1.....	397
A.2	-Qperfinfo=2.....	398
A.3	-Qperfinfo=4.....	400
A.4	-Qperfinfo=8.....	401
A.5	-Qperfinfo=16.....	402
A.6	-Qperfinfo=32.....	404
A.7	-Qperfinfo=64.....	405
A.8	-Qperfinfo=128.....	406
A.9	-Qperfinfo=256.....	407
A.10	-Qperfinfo=512.....	408
A.11	-Qperfinfo=1024.....	409
A.12	-Qperfinfo=2048.....	410
A.13	-Qperfinfo=4096.....	411
A.14	-Qperfinfo=8192.....	412

Figures

1	Microengine Block Diagram.....	15
2	Local Memory Layout.....	48
3	Local Memory Layout for Program 1.....	49

Tables

1	Conventions.....	12
2	Supported CLI Option Switches.....	20
3	Input File Types.....	25



4	Output File Types	25
5	Supported uclid CLI Option Switches	26
6	Summary of Data Types	34
7	Summary of Allowed Combinations of Attributes on Data	52
8	Unaligned Get Functions	75
9	Unaligned Set Functions Summary	78
10	Unaligned memcpy Functions	80
11	Memory I/O Data Types	83
12	Scratch Operation Summary	100
13	SRAM Operations Summary	138
14	DRAM Operations Summary	204
15	MSF Operations Summary	211
16	PCI Operations Summary	222
17	Reflector Operations Summary	227
18	Synchronization Functions Summary	235
19	CSR Access Functions Summary	250
20	Hash Functions Summary	264
21	CAM Access Functions Summary	273
22	CRC Access Functions Summary	281
23	Miscellaneous Functions Summary	310

Introduction

1

1.1 Purpose

This document specifies the subset of the C language supported by the Intel® Microengine C Compiler and the extensions to the language to support the unique features of the Intel® IXP2XXX network processor line.

Note: For simplicity throughout this document, the Microengine C Compiler will be referred to as the *C compiler*, or in some cases simply *the compiler*. Also, the IXP2XXX Network Processor may be referred to as the NPU.

1.2 Features

- The C compiler provides a high-level language programming environment for the network processor to reduce application development time and reduce the need for specialized knowledge.
- The compiler supports programming for the network processor microengines by supporting a combination of the standard C language, language extensions, and intrinsic functions. It supports unique features of the processor through language extensions, intrinsic functions, and inline assembly.
- All existing reference designs currently written in assembly language can be converted to C for the compiler to handle.
- The compiler works with existing tools, including the microcode linker loader and the Developer Workbench.

1.3 Nonfeatures

- The C compiler is not a complete ANSI C implementation.
- There are many features of ANSI C, for example floating point operations, that are outside the realm of applications on the network processor architecture. The compiler omits these features.
- The compiler may not compile existing general purpose C code.
- The compiler does not support the full standard C runtime library.
- It implements useful or necessary functions according to the C runtime library specification, but it does not fully implement the library.
- The compiler does not implement automatic parallelization of code.
- It expects explicitly multithreaded code as input.
- The compiler does not support separate compilation and linking of C code with assembler code.

- The compiler does not support C++.
- The compiler does not support floating point data types (float and double).
- The compiler does not support function pointers and recursion.
- The compiler does not support functions with a variable number of arguments (varargs).

1.4 Conventions Used in this Manual

The following conventions are used in this manual.

Table 1. Conventions

...	Ellipsis indicates that an item may be repeated
[Option]	Items in square brackets are optional
[Option1...]	Optional items can have multiples. The equivalent of [Option1 [Option2]...]
Option=1..5	Range of allowable values. Equivalent to Option=1, 2, 3, 4, or 5
Command1 Command2 Command3	(For Windows*) Selecting cascading options. Indicates that you should follow these steps: Click on Command1 , which offers options including Command2 Click on Command2 , which offers options including Command3 Click on Command3 . For example: Start Programs Accessories Command Prompt
SDK x.y	The x represents the current version, and y the latest point release of SDK that is installed on your system. This could be 3.5, for example
IXP2800 file	Keyboard input, keywords and code items are shown in monospaced font.
IXP2400/IXP2800 PRM [3]	Pointer to another document. In this example, the <i>INTEL® IXP2400 /IXP2800 Programmer's Reference Manual, Chapter 3</i>
IXP2XXX	The family of Intel® IXP2XXX network processors, where 2XXX=the four-digit designator of the target chip.

1.4.1 Version-Specific References

Wherever version-specific files or commands are shown in this book, versions are shown as x and point release numbers are shown as y, as in the following example:

Example: A typical compile command could take this form:

```
ucc1 -Qnctx=1 -I\IXA_SDK_x.y\MicroengineC\include
```

Where x.y = the current software release that is installed on your system.

Overview

2

2.1 Network Processor Architecture Overview

The Intel® IXP2XXX Network Processor contains an Intel XScale® core processor and multiple microengines (8 or 16). This manual is concerned only with compilation of application programs for microengines. Each microengine has hardware support for up to 8 contexts with zero latency task-switches. The microengine can be set up to run 4 contexts instead of 8 by setting a bit in the CTX_ENABLE CSR. In this mode, the 4 contexts that are enabled are context 0, 2, 4, and 6.

For more information on the IXP2XXX Network Processor and the Intel XScale® core processor, refer to the *IXP2400/IXP2800 Programmer's Reference Manual*.

2.1.1 Data Terminology

The following data terminology appears in this document and in the IXP2XXX Network Processor documentation, but not in the Microengine C data types:

Term	Words	Bytes	Bits
Byte	1/2	1	8
Word	1	2	16
Longword	2	4	32
Quadword	4	8	64

Note: [Section 3.1, “Standard Data Types” on page 33](#) details the data types and their naming conventions supported by the C compiler.

2.1.2 Register Model

Each Microengine supports 256 General Purpose Registers (GPR) split into two banks (A and B), and 512 Transfer Registers (XFR) that are used to communicate with memory and I/O devices. The transfer registers are designated as follows:

- 128 SRAM (Scratch) Read XFR registers for I/O
- 128 SRAM (Scratch) Write XFR registers for I/O
- 128 DRAM Read registers
- 128 DRAM Write registers

In addition to these registers, there are also 128 Next Neighbor registers for communication between neighboring microengines. Refer to [Figure 1 on page 15](#) for more information.

The microengines (MEs) support two modes for accessing registers: relative and absolute.

In relative mode, the registers are divided equally between the eight contexts so that each context effectively has its own set of registers. Each context may refer to relative general purpose registers in banks A and B and relative transfer registers without conflicting with the registers of another context.

In absolute mode a context may refer to any of the 256 GPRs. In this mode, some of the registers may be shared among contexts, and others may be context specific.

By setting up a microengine to run in 4-context mode, each context can access twice as many context relative registers. In this mode, odd contexts 1,3, 5, and 7 are disabled and the even contexts 0, 2, 4, and 6 have full access to their registers.

GPRs are located in two separate registers banks (A/B). Only one register from each bank may be read or written in any one clock cycle. Therefore, a typical binary instruction ($w=r0+r1$) may only reference alternating banks for their read operands. The compiler enforces this restriction by potentially adding register moves.

Absolute (ME shared) registers may require extra assembler move instructions as the instruction set is asymmetric (i.e. many instructions do not take absolute registers as operands).

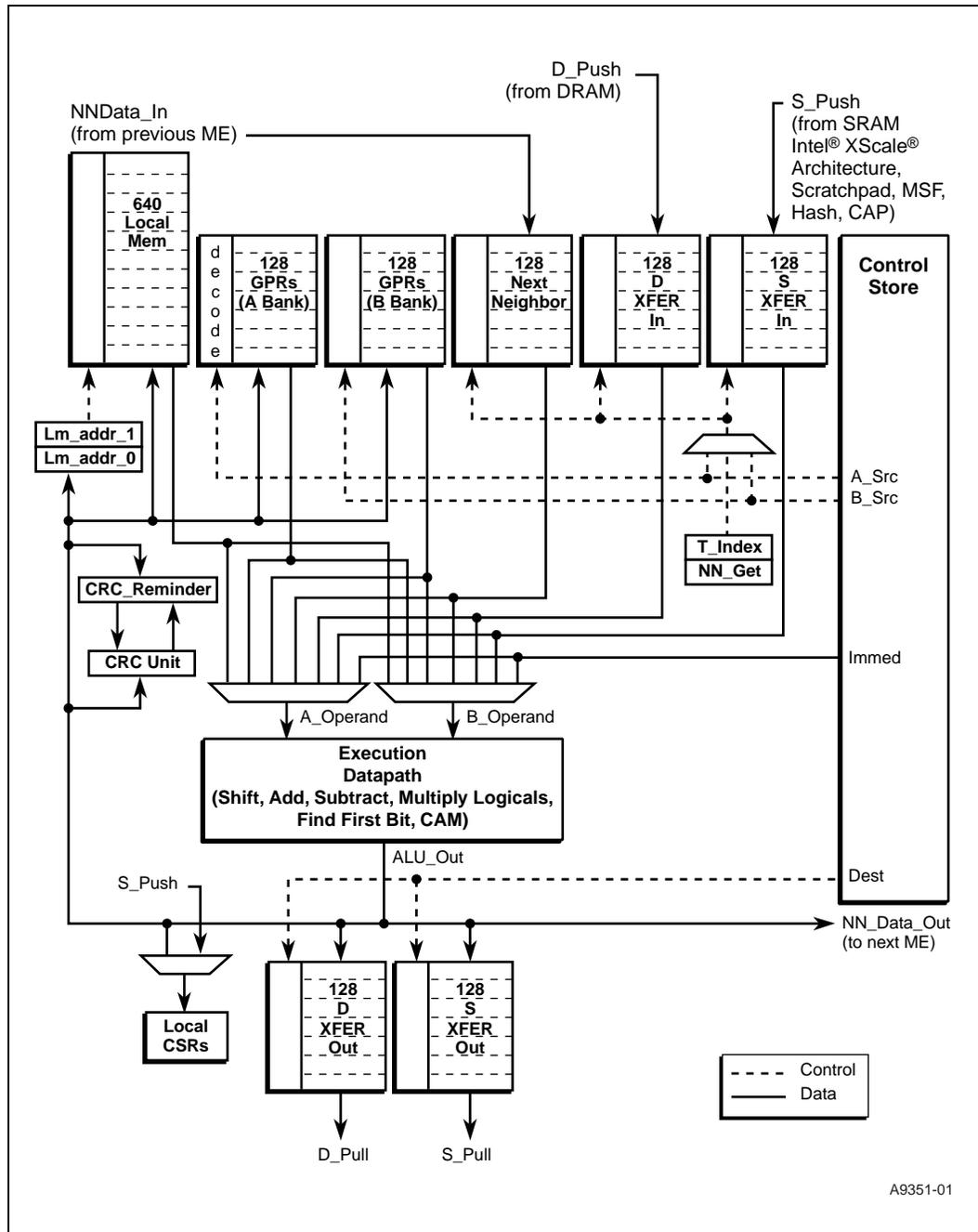
2.1.3 Next Neighbor Registers

Each microengine has 128 next neighbor registers that can be written in one of two ways as selected by the NN_MODE bit in the CTX_ENABLE CSR. When the NN_MODE bit is 0, a write to a next neighbor register goes out of the ME to the corresponding next neighbor register of the next neighbor ME. In this mode, the next neighbor registers of an ME are read-only. When NN_MODE bit is 1, and a next neighbor register is specified as a destination in an instruction, the selected next neighbor register in the same ME is written. When an ME writes to its own Neighbor register, it must wait 5 cycles (or instructions) before it executes the instruction that reads the same register in order to get the newly written value.

Caution: Changing the NN_MODE bit at runtime is not allowed for a Microengine C program and the behavior is undefined.

The next neighbor register can be specified directly as a context-relative register or indirectly through an index register. In the direct access mode the 128 registers are partitioned between the 8 contexts (or 4 contexts in 4-context mode) each addressing its own set (0-15 or 0-31) of context relative registers. In the indirect mode, one of the 128 next neighbor registers is selected through either a local CSR NN_Put ($*n\$index++$ used as destination) or NN_Get ($*n\$index$ or $*n\$index++$ used as source). The compiler supports the indirect access mode through intrinsic functions.

Figure 1. Microengine Block Diagram



2.1.4 Local Memory

Each microengine has a local memory area that is private to it. This memory holds 640 longwords and can be addressed through one of two local memory address CSRs. These CSRs can be configured as either being local to each context or shared among contexts. The local memory can be accessed as operands in microcode instructions (with some restrictions) by setting up one of the two local memory pointers. Hence, it serves as a register set that can be addressed indirectly. There is a 3 cycle latency that must be observed between the setup of the local memory address CSR and its use in dereferencing local memory.

See [Section 3.2.7, “Local Memory Allocation” on page 47](#) for details on local memory layout and allocation.

2.1.5 External Memory

External memory accesses are asynchronous. When memory is read, the thread must do one of the following:

- Swap itself out, allowing other threads to run
- Wait until the operation signals completion before using the data read.

Similarly, when memory is written, care must be taken not to read it or write a new value before the write has completed.

Memory is also divided into three separate regions, each with its own address space. These are:

Memory Region	Speed	Size	Description
DRAM	slowest	largest	directly addressable on quadword boundaries (64-bits) only ¹
SRAM	to	to	directly addressable on longword boundaries (32-bits) only ¹
Scratch	fastest	smallest	directly addressable on longword boundaries (32-bits) only ¹

Notes: 1. The memory is only accessed on a 32-bit (SRAM Scratch) or 64-bit (DRAM) boundary. However, the address specified is a byte address, with the lower 2 bits (3 bits for DRAM) ignored by the memory subsystem.

2. You always read and write memory through transfer registers (XFRs). The transfer registers are divided into read transfer and write transfer registers. The read XFRs are written by external units then used as source, while the write XFRs are read by external units after they are used as a destination. (See [Figure 1 on page 15](#).)

2.1.6 FIFO Queues

In addition to memory, there are two FIFO queues for sending and receiving data. These operate similar to the memory. R_FIFO and T_FIFO refer to RBUF and TBUF respectively. Both can be accessed by either `dram[rbuf_rd,...]`, `dram[tbuf_wr,...]`, or `msf[read,...]`, `msf[read64,...]`,

msf[write,...], msf[write64,...]. Users access them using the above mentioned inline-assembly or intrinsics, or declare variables with `__declspec(tbuf, rbuf, r_fifo, t_fifo)` and use them like C variables.

2.1.7 Signals

The IXP2XXX Network Processor architecture provides a set of 15 signals per context that can be associated with certain hardware events. These signals might be used to notify the execution context that a certain request has been completed. The choice of hardware signal to use is specified in the microcode. Hence these signals are like hardware registers that can be allocated and used by software. Certain events such as DRAM access, or accesses that involve a pull and a push from transfer registers require specification of two signal registers (an even-odd pair) on which the event completion is signaled.

2.1.8 Reflector

The IXP2XXX Network Processor architecture supports an operation called Reflector, which provides the ability for one microengine thread to read its SRAM or DRAM transfer register from the SRAM transfer register or local CSR of another microengine context or write its SRAM transfer register to the SRAM or DRAM transfer register or local CSR of another microengine context. Access completion signals can optionally be requested for one or both of the sending and receiving threads.

2.1.9 Indirect Register Access

The MEv2 architecture supports indirect register access, using the `T_INDEX`, `NN_PUT`, and `NN_GET` registers. Because the value of these registers generally cannot be determined at compile time, the use of indirect register access in inline assembly is not recommended. The compiler will perform register allocation and live range analysis without taking the indirect register access into account, and, as a result, the values read or written using indirect access may be incorrect. For access to the next-neighbor registers in ring mode, you can use the intrinsic functions designed for this purpose. See [Section 4.9, “Miscellaneous Functions” on page 310](#) for additional information.

2.1.10 Threading Model

The programming model for the IXP2XXX Network Processor architecture involves programs running on multiple microengines, each running multiple threads. Each microengine can be configured to run either in 4-context mode or 8-context mode by setting a bit in the `CTX_ENABLE` CSR. In the 4-context mode, twice as many context relative registers are available to each of the 4 threads. The multithreading is explicit; that is, you must partition the tasks across threads. You also need to partition the program across microengines and manage all interthread and interprocess communication.

The IXP2XXX Network Processor is designed to handle a very large number of packets of data in communications routing applications. The threading model contributes to this bandwidth by allowing useful work to be done by another thread while one thread is waiting for completion of a memory or I/O transfer. Thus, the usual case is, when you read or write memory, your thread is swapped out, allowing other tasks to run.

2.1.11 Features Not Supported

Several features generally found on all general-purpose processors are not supported on the IXP2XXX Network Processor microengines. There is:

- No support for either a data stack or a subroutine call stack.
- No data or instruction cache
- No traps or exception support.
- No misalignment support
- No direct support for byte aligned access (direct access must be aligned on longword (SRAM or Scratch) and quadword (DRAM)).

A stack could be implemented in SRAM with software, but due to the long memory latency and multiple instructions required for stack manipulation it would be prohibitively slow. Local memory is not well suited for implementing the stack either, because of its limited size (80 longwords per context). Hence this architecture is not amenable to function recursion and standard caller/callee register partitioning. Consequently, the compiler does not support recursion, and resorts to whole program register allocation to avoid or reduce the overhead of spilling/filling registers.

Function calls are implemented by loading a register with the return address and jumping to the function. The load of the return address is placed in the delay slot of the jump instruction to minimize the overhead.

2.2 Compilation Model

Since microengine programs are necessarily very small, the C compiler always compiles the entire program for a microengine. This is done through the Inter-Procedural Optimization (IPO) feature of the compiler. In this model, you can separately compile C source code, but instead of generating code, the compiler writes the intermediate language to an object file. Then when you link the program, the driver calls the compiler for all of the precompiled objects. The compiler performs a global analysis and calls the code generator for each function in the complete program. At this point, the entire call graph and global usage of all variables is known, allowing for much better code generation than would be possible compiling one function or even one source module at a time.

This compilation model allows the compiler to optimize the linkage between functions based on its knowledge of both the caller and called function. The compiler can allocate static variables to registers, fine tune calling sequences, and avoid the stacking of return addressing and the saving and restoring of registers across calls except when absolutely necessary.

2.2.1 Number of Contexts

When 4 or fewer contexts are needed in the microengine program, it is beneficial to set the microengine to run in 4-context mode. This is done using the compiler switch `-Qnctx_mode=4`. In this mode each context has twice as many context relative registers available, potentially leading to fewer spills and better performance. The exact number of contexts also needs to be specified through a command line option. It is important to remember that in 4 context mode, the contexts are numbered 0, 2, 4, 6. This impacts any code that checks the context number. For example, in 4-context mode the following line of code never evaluates to true:

```
if (__ctx() == 1)
```

2.2.2 Inlining

The compiler performs inlining of functions. This feature of the compiler is controllable through compiler options as well as through the use of directives in the C source code. The keywords `__inline` and `__forceinline` appearing in the function definition (as shown in the example below), indicate to the compiler that the function is to be inlined in all the places it is called from. 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 `-Ob` compiler switch 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.

Example:

```
__forceinline int foo() {...}

__inline int bar() {...}
```

The function to be force inlined with a `__forceinline` keyword should normally be in the same file as the caller to the function. If however the caller and callee are in different files, either a prototype for the function with the "extern" keyword should be present (or included) in the file containing the callee, or the function definition should specify the "extern" keyword in addition to the `__forceinline` keyword.

Example:

```
__forceinline extern int foo() {...}
```

Without this, the force inlined function is treated as if it were a static function and hence is not externally visible or inlined in other files.

The keyword `__noinline` can be used to prevent the compiler from inlining a function. This can be used to control code size.

Note: Use the `-Obn` compiler option switch to control the amount of automatic inlining the compiler will perform. See [Table 2, “Supported CLI Option Switches” on page 20](#) for more information on the `-Obn` compiler switch.

2.3 Running the C Compiler

You can compile your C source code in one of two ways, by using one of the following:

- Compiler command line interface (CLI)
- Developer Workbench

2.3.1 The Command Line

You can use the compiler command line interface from a command prompt window on your system. Do the following:

1. Open a command prompt window.
2. Go to the folder containing the C source files, typically:

```
C:\IXA_SDK_3.y\me_tools\bin>
```

3. Invoke the C compiler using this command:

```
uccl [options] filename...
```

Note: C:\IXA_SDK_3.y\me_tools\bin should be on your PATH. On a system with IXA SDK 2.y and IXA SDK 3.y installed, the Micro C compiler runs on either the V2.y or V3.y DevWorkbench. If you want to run the V3.y C compiler from the command line, the system path needs to be set up accordingly.

2.3.2 Supported Compilations

Two kinds of compilations are supported:

- Compile one or more source files (*.c, *.i) into separate object (*.obj) files.
- Compile any combinations of source file (*.c, *.i) and/or object file (*.obj) into one list file (*.list).

In the first case, you must use the `-c` switch in the command line in order to compile .c files into separate .obj files. You might want to use this method to compile .c files that don't change very often, for example, rtl.c, so that you don't have to recompile them every time you make a .list file.

Example: `uccl -c file1.c file2.i`

In the second case, do not use the `-c` switch. In the following example, two source files (.c and .i) and an object file (*.obj) are compiled to produce a .list file.

Example: `uccl file1.c file2.i rtl.obj`

2.3.3 Supported Compiler Option Switches

Table 2 lists and defines all the supported C compiler command line switches. The Command Line Interface (CLI) warns and ignores unknown options. The CLI honors the last option if it conflicts with a previous one, for example,

```
uccl -c -O1 -O2 file.c
```

this generates the following warnings and proceeds:

```
uccl: Command line warning: overriding '-O1' with '-O2'
```

If you enter other conflicting switches such as `-E` and `-EP`, the last switch entered always prevails.

Options that do not take a value argument, such as `-E`, `-c`, etc., are off by default and are enabled only if specified on the command line.

Table 2. Supported CLI Option Switches (Sheet 1 of 5)

Switch	Definition
-? -help	Lists all the available options.
-c	Compiles each .c or .i file to a .obj file (rather than compile and link).
-Dname[=value]	Specifies a #define symbol. The value, if omitted is 1.

Table 2. Supported CLI Option Switches (Continued) (Sheet 2 of 5)

Switch	Definition
-DSDK_3_y_COMPATIBLE	Uses the IXA SDK 3.y version of the hash intrinsics (with the read and write parameters swapped) and removes error checking for generic (“void **”) typecasts in intrinsics library parameters. If possible, SDK 3.y code should be changed to work with the new versions of the hash intrinsics and any generic typecasts should be changed to the correct types.
-E -EP -P	Preprocess to stdout. Preprocess to stdout, omitting #line directives. Preprocess to file.
-Fa<filename>	Produces a .uc file containing the generated microcode intermixed with the source program lines. The resulting assembly file is for reference only; the compiler does not guarantee that the file will pass through the assembler. If an assembler-compatible file is required, the -uc option should be used instead. This may have a negative impact on performance, however; certain optimizations cannot be performed when compiling for the assembler.
-Fo<file> -Fo<Dir\>	Name of object file or directory for multiple files.
-Fe<file>	Base name of executable (.list, .ind) file. Defaults to the base name of the first file (source or object) specified on the command line followed by the extension (.list).
-Fi<file>	Overrides the base name of the .ind file.
-FI<file>	Forces inclusion of file.
-Gx2XXX (where 2XXX=the target NPU model)	Specifies the target processor. IXP2800 is the default. The compiler adds -DIXP2XXX as appropriate.
-I path[:path2...]	Path(s) to include files, prepended before path(s) specified in environment variable UCC_INCLUDE.
-link[linker options]	Calls the microengine image linker (uclid) after successful compilation, passing any specified linker options. The default linker options are: -u 0 -sc 0x00000004:0x00003ff4 -dr 0x00000010:0x07ffffe8 -sr0 0x00000004:0x03ffffc -sr1 0x00000004:0x03ffffc -sr2 0x00000004:0x03ffffc -sr3 0x00000004:0x03ffffc”
-Obn	Inlining control: n=0, none; n=1, explicit (inline functions declared with <code>__inline</code> or <code>__forceinline</code> (default)); n=2, any (inline functions based on compiler heuristics, and those declared with <code>__inline</code> or <code>__forceinline</code>)
-On	Optimize for: n=1, size (default); n=2, speed; n=d, debug (turns off optimizations and inlining, overriding -Obn below).
-Qbigendian	Compile big-endian byte order (default). Compiler adds -DBIGENDIAN, -ULITTLEENDIAN. All other command line BIGENDIAN/ LITTLEENDIAN symbol definitions and undefinitions are ignored.
-Qdefault_sr_channel=<0...3>	Specify the SRAM channel that should be used when allocating compiler-generated SRAM variables and variables that are specified as <code>__declspec(sram)</code> . The default is channel 0.
-Qerrata	Report when the compiler-generated code triggers a known processor erratum.

Table 2. Supported CLI Option Switches (Continued) (Sheet 3 of 5)

Switch	Definition
-Qip_no_inlining	Turns off all inter-procedural inlining. Inter-procedural inlining is on by default.
-Qlittleendian	Compiles little endian byte order. Compiler adds -DLITTLEENDIAN -UGIBIGENDIAN. All other command line LITTLEENDIAN/BIGENDIAN symbol definitions and undefinitions are ignored.
-Qliveinfo	Equivalent to -Qliveinfo=all
-Qliveinfo=gr,sr,...	Print detailed liveness information for a given set of register classes: gr: general purpose registers sr: SRAM read registers sw: SRAM write registers srw: SRAM read/write registers dr: DRAM read registers dw: DRAM write registers drw: DRAM read/write registers nn: neighbor registers (only when -Qnn_mode=1) sig: signals all: all of the above
-Qlm_start=<n>	Provides a means for user to reserve local memory address [0, n-1] (in longwords) for direct use in inline assembly. Compiler does not allocate any variables to this address range.
-Qlm_unsafe_addr	Disables the compiler's use of local memory auto increment addressing. Used when user code writes local memory pointers with invalid values. See Section 6.2.6 for more information.
-Qlmpt_reserve	Reserve local memory base pointer I\$index1 for user inline assembly code.
-Qmapvr	Prints out pseudo-assembly code with annotations that map physical registers to user variables and compiler-generated temporary variables.
-Qnctx=<1, 2, 3, 4, 5, 6, 7, 8>	Specifies the number of contexts that will be made active in your program. Unused contexts will be made to execute the <code>ctx_arb[kill]</code> instruction and terminate. Compiler-allocated resources such as memory will not be allocated to unused threads. The underlying number of contexts supported by the hardware will not be changed, so hardware-managed resources such as registers will still be allocated to all threads. Defaults to the value of <code>-Qnctx_mode</code> (which defaults to 4). If <code>-Qnctx</code> is set greater than the value of <code>-Qnctx_mode</code> , <code>-Qnctx_mode</code> will be changed to the higher value.
-Qnctx_mode=<4, 8>	Specifies the number of contexts that the hardware will support. Changing this value from 4 to 8 halves the number of available context-specific registers. Defaults to 4.
-Qnn_mode=<0, 1>	Sets <code>NN_MODE</code> in <code>CTX_ENABLE</code> for setting up next neighbor access mode. (See Next Neighbor Register section in Chapter 3). 0=neighbor (default), 1 = self).
-Qnolur=<func_name>	Turns off loop unrolling on specified functions. You can supply one or more function names to the option. For example: -Qnolur="_main"; turn off loop unrolling for main(). -Qnolur="_main,_foo"; turn off loop unrolling for main() and foo(). The supplied function name must have the preceding underscore ('_').
-Qold_revision_scheme	Generates hardware revision numbers that are compatible with IXA SDK 3.0 and below.

Table 2. Supported CLI Option Switches (Continued) (Sheet 4 of 5)

Switch	Definition
-Qperinfo=n	<p>Prints performance information.</p> <p>n=0 - No information (similar to not specifying)</p> <p>n=1 - Register candidates spilled (not allocated to registers) and the spill type</p> <p>n=2 - Instruction-level symbol liveness and register allocation</p> <p>n=4 - <deprecated></p> <p>n=8 - Function sizes</p> <p>n=16 - Local memory allocation</p> <p>n=32 - Live range conflicts causing SRAM spills</p> <p>n=64 - Instruction scheduling statistics</p> <p>n=128 - Warn if the compiler cannot determine the size of a memory I/O transfer</p> <p>n=256 - Display information for "restrict" pointer violations</p> <p>n=512 - Print offsets of potential jump[] targets</p> <p>n=1024 - Information about the Boolean propagation optimization</p> <p>n=2048 - Register requirements report</p> <p>n=4096 - Information on switch statement optimizations</p> <p>n=8192 - Print information on I/O parallelization</p>
-Qrevision_min=n -Qrevision_max=m	<p>The version arguments allow the compiler to generate code that works on a range of processor versions (steppings).</p> <p>0x00=A0 (default for -Qrevision_min)</p> <p>0x01=A1</p> <p>0x10=B0</p> <p>0x11=B1</p> <p>The default revision range is 0x00 to 0xff (all possible processor versions). The default for -Qrevision_max is 0xff. The compiler adds -D__REVISION_MIN=n and -D__REVISION_MAX=m. Note: The IXP program loader reports an error if a program compiled for a specific set of processors is loaded onto the wrong processor.</p>
-Qspill=<n>	<p>Selects the alternative storage areas ("spill regions") chosen when variables cannot be allocated to general-purpose or transfer registers:</p> <p>(LM=local memory, NN=next neighbor registers)</p> <p>n=0: LM (most preferred) -> NN -> SRAM (least preferred)</p> <p>n=1: NN->LM->SRAM</p> <p>n=2: NN only; halt if not enough NN</p> <p>n=3: LM only; halt if not enough LM</p> <p>n=4: NN->LM; halt if not enough LM or NN</p> <p>n=5: LM->NN; halt if not enough LM or NN</p> <p>n=6: SRAM only</p> <p>n=7: No spill; halt if any spilling required</p> <p>n=8: LM->SRAM</p> <p>Default is n=0. You must set -Qnn_mode=1 to use the NN registers as a spill region. If the NN registers are used by program code, NN spilling will be automatically disabled.</p>
-s	<p>Changes the behavior of -uc by not calling uca to assemble the compiler produced assembly code. Only valid when combined with -uc option.</p>

Table 2. Supported CLI Option Switches (Continued) (Sheet 5 of 5)

Switch	Definition
-uc	Mixing C and microcode programming. Under this option, you can compile one or more C files as well as one or more microcode files into one application. The compiler compiles all C files into one microcode file, then sends this microcode file as well as other microcode files to UCA to produce a list file.
-Wn n=<0, 1, 2, 3, 4>	Warning level. 0=print only errors 1, 2, 3=print only errors and warnings 4=print errors, warnings, and remarks. Defaults to 1.
-Zi	Produces debug information. The compiler generates a file with a .dbg extension for each source.

2.3.3.1 Environment Variables

The following environment variable is recognized by the compiler:

UCC_INCLUDE: A list of directories to be added to the include path. The list is separated by semicolons: dir1;dir2;dir3..., and is appended after the directories supplied on the command line using -I.

2.3.4 Input and Output File Types

Table 3. Input File Types

Extension	File type
.c	source file
.h	header file
.i	source file after preprocessing
.obj	object generated by the compiler invoked by the -c switch

Table 4. Output File Types

Extension	File type	Command switch
.list	output file from compiler used by linker	-Fe<file>
.obj	object generated by the compiler invoked by the -c switch	
.uc	assembler input generated by the compiler invoked by the -Fa switch	
.uof	linker output file	
.ind	a Transactor script to assemble and run the program	-Fe<file> or -Fi<file>

2.3.5 Linking a Microengine .UOF file

Before running a program you must link using the ucl linker. The linker can combine multiple files into a single executable .uof file. The executable file can contain microcode and data for one or more microengines.

Example: To link a single microengine program to run on microengine 0 use:

```
ucl -u 0 -dr 0x00000010:0x07ffffe8 file.list
```

Example: Using the -link switch:

```
uccl file.c -link
```

You can override linker options after -link. The default supplied by uccl is:

```
-u 0 -sc 4:0x00003ff4 -dr 0x10:0x07ffffe8 -sr0 4:0x03fffffc -sr1 4:0x03fffffc  
-sr2 4:0x03fffffc -sr3 4:0x03fffffc
```

Table 5 lists and defines all the supported C linker command line switches. The CLI ignores and issues a warning for unknown options.

Table 5. Supported uclid CLI Option Switches

Switch	Definition
-u	Specifies the Microengine the program is meant for.
-sr0, -sr1, -sr2, -sr3,	Specifies location in SRAM memory (channel 0, 1, 2, 3) to allocate variables.
-sc	Specifies location in Scratch memory to allocate variables.
-dr	Specifies location in DRAM memory to allocate variables.

You may want the .list file to contain special linker directives that are not directly supported by the compiler. For example, the “.%image_name” directive is used by the linker to tag the .uof file with a text label. In this case, you can use the #pragma comment(linker...) directive as follows:

```
#pragma comment(linker ".%linkerdirective arg1 arg2")
```

This directive will emit “.%linkerdirective arg1 arg2” directly at the top of the .list file.

2.3.6 Util.c

The util.c file provides functions that can be used to display characters, strings, and numbers in various formats. These functions use the put() intrinsic to simulate output of a single character. The put function makes use of Scratch memory 0x3ff8-0x3fff. Output is simulated by setting a watch point in the simulation script file (.ind) on Scratch memory 0x3ff8 and printing data in longword 0x3ffc.

2.3.6.1 Utility Functions (util.c)

The util.c resides in the MicroengineC\Samples\util underneath the IXA SDK installation directory. It contains a collection of functions that can be used to display characters, strings, and numbers in various formats. The functions available are:

void put(int c)	Puts a single character.
void puts(char *a)	Puts a string of characters in SRAM. A quoted string can be used as the argument, for example, puts("Hello world\n");
void putui(unsigned int x)	Formats and puts an unsigned int in decimal.
void putsi(int x)	Formats and puts out a signed int in decimal.
void putull(unsigned long long x)	Formats and puts out an unsigned long long in decimal.
void putsll(long long x)	Formats and puts out a signed long long in decimal.
void puthi(int x)	Formats and puts out an unsigned int in hex.

```
void puthll(unsigned long long x)  Formats and puts out an unsigned long long in
                                hex.
```

If you run a program through the Transactor from the command line, use the .ind file generated by the compiler to run the program. The output will appear in your **Command Line** window.

If you run from the Workbench, the util.ind script file in the util directory needs to be included in your project in order to have the output appear in the **Command Line** window.

Note: This script should be included as an additional initialization script in the project.

2.3.6.2 Multi-threading restrictions

All of these functions are thread-safe at the character level (that is, multiple threads can be writing characters without interfering with each other). But if you have multiple threads writing, for instance a number, the digits will get intermixed as thread swaps occur. To avoid this, use a thread lock mechanism to control how output from multiple threads are intermixed. Refer to [Chapter 8, “Mutual Exclusion Library”](#), for information on appropriate locking mechanisms.

The following facilitates multiple I/O data types in a Microengine thread-safe manner, similar to printf. In these examples, the IO_ macros are defined in util.h. The pm_printf() function is define in util.c.

```
io_item  io[2];

IO_ADD_STRING(io[0], "This is a test");
IO_ADD_INT(io[1], 123);
pm_printf(2, io_dlmtr_nl, io);

IO_ADD_CHAR(x,v) // add char to print
IO_ADD_UINT(x,v) // add unsigned int
IO_ADD_INT(x,v)  // add int
IO_ADD_ULL(x,v)  // unsigned long long
IO_ADD_LL(x,v)  // long long
IO_ADD_ULLH(x,v) // unsigned long long in hex
IO_ADD_INTH(x,v) // int in hex
IO_ADD_STRING(x,v) // string
IO_ADD_PTR(x,v)  // generic pointer
```

2.3.7 Example—Using the C Compiler

A simple C program, hello.c, displays “Hello World” on the screen.

2.3.7.1 The C File

Hello.c looks like this:

```
#include <util.h>
void main()
{
    puts("Hello world\n");
}
```

2.3.7.2 Compiling the File

To compile, the command line looks like this:

```
ucc1 -Qnctx=1 -I\IXA_SDK_3.y\MicroengineC\include \
      -I\IXA_SDK_3.y\MicroengineC\samples\util hello.c \
      \IXA_SDK_3.y\MicroengineC\samples\util\util.c \
      \IXA_SDK_3.y\MicroengineC\src\rtl.c \
      \IXA_SDK_3.y\MicroengineC\src\intrinsic.c
```

or if util and rtl are pre-compiled:

```
ucc1 -Qnctx=1 -I\IXA_SDK_3.y\MicroengineC\include \
      -I\IXA_SDK_3.y\MicroengineC\samples\util hello.c \
      \IXA_SDK_3.y\MicroengineC\lib\util.obj \
      \IXA_SDK_3.y\MicroengineC\lib\rtl.obj \
      \IXA_SDK_3.y\MicroengineC\lib\intrinsic.obj
```

Note: Use the slash character (/) in place of the backslash character (\) as the directory delimiter under the Linux operating system.

2.3.7.3 Linking the File

To link, enter the following command on the command line:

```
ucc1d -u 0 -dr 0x00000000:0x80000000 hello.list
```

Note: You can skip this step by specifying the `-link` switch during compilation.

2.3.7.4 Running the File

The `hello.ind` file is generated during linking. To run the hello world program, type the following command:

```
IXP2800 < hello.ind
```

or

```
IXP2400 < hello.ind
```

The system displays a screenful of text from the Transactor initialization, followed by the text “Hello world” and a count of the cycles it took to execute.

For more extensive example code, refer to `rtl.c` and `util.c`.

2.3.7.5 Initialization File

The initialization (script) file `util.ind` resides in `\MicroengineC\samples\util` and is used with the Developer Workbench for simulating output to `stdout`.

2.3.8 C Compiler Graphical User Interface from Developer Workbench

The Developer Workbench supports an integrated microengine C compiler for creating, compiling, testing, and debugging Microengine C applications.

2.3.8.1 Build Features

The Developer Workbench supports the following Microengine C build features:

- Support for creating, editing, and managing C source files.
 - Inserting C sources into project.
 - Dependency checking for .c and .h files.
 - Syntax coloring for .c and .h files.
 - FileView for .c files.
- Support for setting up projects to include producing microstore images from C source files and Assembler .uc files.
 - Build setting/compiler dialog.
 - GUI control for specifying compile options.
 - Support for include paths for C sources separate from Assembler include paths.
 - Support for setting target paths for images.
 - GUI for preprocessor symbol definition to support #ifdef.
 - GUI controls for specifying link options.
- Support for building microstore images from C source files and Assembler .uc files.
 - Running compiler and providing parameters.
 - Displaying and interpreting output messages.
 - Relating errors and warnings to source lines.
- Support for persisting project and option files.
 - Persist new project data in dwp file.
 - Persist new option data in dwo file.

2.3.8.2 Debug Feature

The compiler supports source level debugging. However, when compiling for debug, optimizations are generally turned off.

- Support for source-level debugging.
 - Display of register and variable values based on scope and live-range.
 - Display of memory variables.
 - Display of values as C structures.
 - Optional expanded display of assembly instructions generated by each C source statement.

- Single-stepping based on C source statements or expanded assembly instructions.
- Setting breakpoints.
- Run to cursor.
- Current instruction markers.
- Execution coverage.
- Thread history.
- Data watches.
- Go to source.

2.4 Running and Debugging Under the Developer Workbench

To create and run a project under the Developer Workbench, perform the following steps:

1. Start the Developer Workbench.
Depending on how you installed it, you can usually click **Start** on the Task bar and then select **Programs-> Intel IXA_SDK_3.y->Developer Workbench**.
2. On the **File** menu, click **New Project**.
3. Enter the location and project name, select chip type IXP2400 or IXP2800, then click **OK**.
4. On the **Project** menu, click **Insert compiler source file**. Normally, you need to insert `rtl.c` and `intrinsic.c` if want to use intrinsic functions, `util.c` if you include `rtl.c` or want to use character I/O, and `libc.c` if you want to use string operations. These files are included from the `MicroengineC\src` and `MicroengineC\samples\util` directories.
5. Select the `.c` file(s) needed for your project and click **Insert**.
6. On the **Build** menu, select **Settings**.
7. Click the **C Compiler** tab.
8. Click **New** to specify the path and name of the output (`.list`) file.
9. Click **Choose source files**.
10. Select the needed source files for this program.
11. Click the **Linker** tab.
12. Click the browse button to the left of the **Output to target .uof file** box.
13. Enter the name of the `.uof` file you wish to create.
14. From the list under **Select files for microengine 0** and select the file you specified in Step 8 and then click **OK**.
15. Click the **General** tab and add the compiler include directories. Normally, you need to include the `C:\IXA_SDK_3.y\MicroengineC\include` and the `C:\IXA_SDK_3.y\MicroengineC\samples\util` directories.
16. On the **Build** menu, click **Build**.
17. If you get no errors, you can start debugging by selecting **Start Debugging** on the **Debug** menu.

Note: The compiler I/O functions provided in MicroengineC\samples\util\util.c such as puts(), etc. operate by writing to Scratch memory 0x3ffc-0x3fff. This register is also used to signal exit from function main(). To see this generated output in a simulation session, or to halt simulation on exit from the main() program, a watch on Scratch memory needs to be included (an example is provided in MicroengineC\samples\util\util.ind). Note that when used in Workbench the register name in the .ind file must be prepended with the name of the chip (or nothing if no chip name is specified).



C Language Support

3

3.1 Standard Data Types

3.1.1 Basic Data Types

The compiler supports the following standard scalar data types:

- char 8-bit signed and unsigned
- short 16-bit signed and unsigned
- int 32-bit signed and unsigned
- long 32-bit signed and unsigned
- long long 64-bit signed and unsigned
- enum 32-bit signed and unsigned
- pointers 32-bit pointers typed by memory type

As per the C standard, chars are the smallest addressable units, and pointers to successive chars differ by one. The compiler supports chars and shorts and pointers to them, although at some potential performance cost. Users are recommended to avoid usage of char and short when possible, because access of quantities less than 32 bits (64 bits in DRAM) generally involves additional operations to extract the appropriate bytes from the longword or quadword. Access through pointers to 8-bit and 16-bit types may also require runtime alignment of data, which is even more inefficient.

3.1.2 Pointer Representation

All pointers are represented as byte addresses irrespective of the memory region pointed to. The compiler keeps track of the memory region that a pointer can point to and issues error messages on inconsistent use. For example, assignment of a pointer to SRAM to a pointer to DRAM will be flagged as a user error. Pointers with no specified memory region are assumed to point to SRAM. Further, when performing pointer arithmetic, the compiler will modify the byte value by the appropriate value. For example, when incrementing a pointer to a long long by one, the compiler will add 8 to the pointer value. If the pointer is pointing to a user defined data type, the compiler will also do the right thing. See [Section 3.1.9](#) for more information on how the compiler handles alignment issues.

3.1.3 Bitfields

The compiler implements arrays and structs. Since the Intel® IXP2XXX processors handle packets of communication data that are often defined in terms of bit fields, the compiler supports efficient manipulation of structs with bit fields. Bit fields are supported using standard C syntax. The compiler also supports packed bit-fields through `__declspec(packed_bits)` as described in [Section 3.1.7](#). With this declspec, structures containing bitfields are laid out such that there is never

any padding inserted between a bit-field and its previous member in the structure. `__declspec(packed_bits)` is helpful in mapping packet header structures accurately onto C structures. See [Section 3.1.7](#) for more information on packing bit fields within a structure.

3.1.4 Floating Point Types

The compiler does not support floating point types. The lack of hardware support and limited code space make it virtually impossible to provide any floating point support-nor is any needed for the type of applications envisioned.

3.1.5 String Literals

String literals are placed into SRAM and accessed through a pointer to SRAM. It is an error to use a string literal in a position which expects a pointer to a non-SRAM memory region, unless a static initialization of a character array is being performed. Example:

```
void foo(__declspec(dram) char *str_in_dram) { ... }

foo("string"); // ERROR: "string" is in SRAM and cannot be passed to foo()

foo((__declspec(dram) char *)"string"); // RUNTIME ERROR: address of "string" is
not a valid DRAM address
{
    __declspec(dram) char *ptr = "string"; // ERROR: "ptr" must be a character
array
    __declspec(dram) char arr[7] = "string"; // CORRECT: static initialization of
character array
    foo(arr); // CORRECT: type of parameter matches
type of argument
}
```

3.1.6 Size of Data Types

The size of data types, as reported by the `sizeof()` built-in function, are in bytes, thus:

```
sizeof(int) == 4
sizeof(long long) == 8
```

[Table 6](#) lists the standard C built-in data types and indicates which are supported by the Microengine C compiler.

Table 6. Summary of Data Types (Sheet 1 of 2)

Data Type	Supported	Size (in bits)
char	Yes	8
short	Yes	16
int	Yes	32
long	Yes	32

Table 6. Summary of Data Types (Continued) (Sheet 2 of 2)

Data Type	Supported	Size (in bits)
long long	yes	64
enum	Yes	32
pointers	Yes	32
float	No	N/A
double	No	N/A
struct	Yes	Variable
union	Yes	Variable
array	Yes	Variable

3.1.7 Alignment of Data Types

The compiler aligns all chars on 1 byte boundaries, shorts on 2 byte boundaries, int, enum, long, and pointer data types on a 4 byte boundary, and all long long data types on an 8 byte boundary. In general, pointer values can be assumed to be aligned based on the type of the data pointed to. The compiler inserts padding between elements of structures to ensure that each element is aligned based on its type unless the structure has been declared with the `__declspec(pack)` qualifier. (See [Section 3.1.8, “Packed Aggregates” on page 36](#) for more information.)

Bit fields are stored within longwords. The next bit field starts at the next bit position within the current longword if, and only if, the entire bit-field element fits within the current longword (4 bytes). If the bitfield is too wide to fit, then the remaining bits in the current longword are padded and the bit-field begins at the next longword. This bitfield padding is avoided, however, if the structure is declared with a `__declspec(pack, bits)`. In this case, the overflow bits of the bitfield wrap to the next longword and the bitfield is split between two longwords.

An aggregate (array, union, structure) assumes the strictest alignment of any of its members. Hence it is aligned on a 8 byte boundary if the aggregate contains a long long member; otherwise, it is aligned on a 4 byte boundary if it contains an int or long member, etc. A final tail padding is added to each structure to make its size a multiple of its required alignment. This guarantees that when you have an array of structs no additional padding is needed between array elements to align all the element structs. Aggregates (structures/unions/arrays) allocated explicitly in SRAM/Scratch/local memory are additionally aligned at least on a 4 byte boundary. Similarly, aggregates allocated explicitly in DRAM are aligned at least on an 8 byte boundary.

The alignment of a given structure can be changed with the `__declspec(aligned(n))` directive, where “n” is a power of two, up to 2048 for memory and 64 for local memory. If the structure's natural alignment is less than the word size of the structure's storage region (16 for DRAM, 4 for other types of storage), the performance of whole structure copies can be improved by increasing the alignment value (padding) to the word size. In the following example the natural alignment is 1, but it can be changed to 4 by using the align directive.

```
typedef __declspec(sram, aligned(4)) struct // overrides natural alignment
{
    char c; // natural alignment is "1" because of this element
    char s;
```

```

} str;

...

str x,y;

...

x = y; // copy performance improved by manually setting alignment

```

Note: `__declspec(...)` qualifiers must be placed to the left of the “struct” keyword, as in the example above.

Structure alignment is not optimized automatically because of the possibility that the structure may be embedded inside an array or another structure, which calls for the use of the structure's natural alignment.

If a structure is allocated in NPU local memory, setting the alignment to the closest power of two greater than the size of the structure (see [Section 3.2.7.5, “Alignment Information for Local Memory Pointers” on page 50](#)) will allow the compiler to generate faster, offset-based addressing for the structure members. The disadvantage of doing this is an increase in the amount of wasted space needed to pad such objects.

Note: SRAM and DRAM support access on longword or quadword granularity respectively. Extraction or modification of bytes within a longword or quadword involves generation of additional instructions, and consequently results in some performance degradation.

3.1.8 Packed Aggregates

Aggregates (structures, unions, and arrays) are normally aligned on byte boundaries. Padding (up to 64 bytes) of aggregates is an automatic compiler function to improve performance. (This is discussed in [Section 3.1.7.](#)) Under some conditions you may wish to block padding. A `__declspec(packed)` qualifier can be used to avoid the padding between members of the aggregate and to avoid tail padding.

Note that `__declspec(packed)` implies `__declspec(packed_bits)`.

The naturally aligned data for an NPU for Scratch and SRAM is 4 bytes; for DRAM on IXP2400 it is 4 bytes, for IXP2800 it is 8 bytes. SRAM and Scratch rings require alignment to 512 bytes.

Local memory natural alignment is 4 bytes, but when referencing big structures, the performance will improve with bigger alignment (up to 64 bytes). The reason is the hardware uses an "OR" to calculate the local memory address. For example, when you use `*(p+16)`, assuming that `p` is your base pointer; if `p` is aligned to 16, then `p|16` (`p OR 16`) is the same as `p+16`. If `p`'s alignment is smaller than 16, we have to perform an ADD and reset the local mem base pointer, which is very time consuming.

3.1.9 Pointer Alignment Assumptions and Unaligned Pointers

Normally declared pointers will be assumed to point to data with correct alignment based on the natural alignment of the type they point to. The compiler generates code that correctly dereferences these pointers only in the case that they are correctly aligned. If they are not correctly aligned, the behavior is undefined.

Example:

```
char *pc;      // pc can have any byte address
int *pi;      // pi must be zero mod 4.
short *ps;    // ps must be zero mod 2.
```

Additional alignment assumptions are made on values of variables declared as pointers to aggregates in memory. By default variables declared as pointers to aggregates allocated to SRAM, Scratch, or local memory are assumed to point to objects with a 4-byte minimum alignment. Variables declared as pointers to aggregates allocated to DRAM are assumed to point to objects with an 8-byte minimum alignment.

By using `__declspec(unaligned)` all alignment assumptions on the value of the pointer is avoided and the compiler will generate correct code to dereference the pointer. This code is considerably larger / slower than the code for aligned pointers.

Any pointer to a component of a packed aggregate is an unaligned pointer. An unaligned pointer cannot be assigned to an aligned pointer of the same type. Hence pointers to packed variables can be assigned only to unaligned pointer variables but not to regular pointer variables. An aligned pointer can be assigned to an unaligned pointer of the same type, since it is less restrictive.

Example:

```
int *pi;
__declspec(unaligned) int *pui;
__declspec(packed) struct {char x; int y} z;

pi = &z.y;      // Error since &z.y is an unaligned pointer
pui = &z.y;    // Ok since pui is an unaligned pointer
```

In this example, “pui” can point to any byte address, whereas “pi” can only point to 4-byte-aligned addresses.

The `__declspec(aligned(n))` attribute can be used to declare the alignment of the objects that a given pointer will reference. “n” must be a byte value and a power of two. If you know that a pointer that is normally unaligned (a pointer to char, for instance) will only reference objects that are aligned on word boundaries, declaring the pointer with a higher alignment (`__declspec(aligned(4))` for SRAM, `__declspec(aligned(8))` for DRAM) will allow the compiler to generate faster code when dereferencing such pointers.

For example:

```
__declspec(packed) struct // structure is 4 bytes long but 1-byte aligned
{
    char a, b;
    short c;
} *ptr;
...
ptr x, y;
...
*x = *y; // unaligned copy
```

Since the natural alignment of this struct is 1 (single-byte aligned), the compiler will make no assumptions about the position of the structure in memory, and will generate code for the copy operation that will take into account the fact that the structure might span across two memory words, taking up only part of each word. If `__declspec(aligned(4))` is added to the structure definition:

```
__declspec(packed, aligned(4)) struct
{
    char a, b;
    short c;
} *ptr;
...
ptr x, y;
...
*x = *y; // optimized copy
```

The compiler now assumes that the structure fits entirely into a single word in memory, and generates code that reads and writes only a single word.

3.1.10 Endian Support

Since the processor supports both big-endian and little-endian applications, and since internet data structures are typically laid out big-endian, the compiler supports both big-endian and little-endian layout of aggregates. A compiler switch selects the endian mode. (Refer to [Table 2, “Supported CLI Option Switches”](#) on page 20 for more information.)

3.1.10.1 Compiler Limitations of Endian Support.

There are several cases in which the compiler cannot compensate for inherent little-endian bias in the IXP2XXX microengine hardware. These are:

- [Section 3.1.10.1.1, “Hash Instructions and Related Intrinsics”](#) on page 39.
- [Section 3.1.10.1.2, “DRAM Partial Writes”](#) on page 39.

The following sections have more information on these issues.

3.1.10.1.1 Hash Instructions and Related Intrinsic

The hash instructions compute a polynomial of input data and hash multipliers on 48, 64, or 128 bit quantities. The hardware treats the input data and output data as little-endian, that is, if a 48 bit quantity is used, and the data is in xfer register \$0 and \$1, the low 32 bits are in \$0 and the high 16 bits are in \$1. If you use a long long to represent this data, it is correct only in little-endian mode. In big-endian mode, you have to swap the two longwords of data before the hash and after the hash to get equivalent results.

The compiler cannot do this swapping automatically, because the hash operation are always asynchronous (i.e. sig_done is required), and the compiler does not always know when the operation is finished. For information on hash operations, refer to the *IXP2400/IXP2800 Programmer's Reference Manual*.

Example: To do a hash of a 48 bit number in big-endian mode, where bit 0 is the low order bit, do the following:

```
typedef union
{
    struct
    {
        int lo, hi;
    } s;

    long long ll;
} lw;

lw in, out, xfer_in, xfer_out;
SIGNAL_PAIR sp;

/* swap the long long into the xfer register buffer */
xfer_in.s.lo = in.s.hi;
xfer_in.s.hi = in.s.lo;

/* hash it into the xfer register out buffer */
hash_48(&xfer_in, &xfer_out, 1, ctx_swap, &sp);

/* swap back into the out */
out.s.hi = xfer_out.s.lo;
out.s.lo = xfer_out.s.hi;
```

3.1.10.1.2 DRAM Partial Writes

The DRAM instruction allows you to write any set of the 8 bytes in a quadword. The bytes to be written are specified by a byte mask.

Example: To write the low byte only, you specify binary 00000001. However, this only writes the lowest order 8 bits of a long long in little-endian mode. In big-endian mode this writes bits 39:32. To write bits 7:0 in big-endian mode, you have to use the mask 00010000. Essentially, the byte positions within a longword are correct in either mode, but the two longwords are reversed in big-endian mode.

Example: This example sets the lowest 8 bits in DRAM to the value in BIGENDIAN mode. It sets the variable “mem” to be the hex value 0x00000000 000000ef.

```
#include <ixp.h>
void main () {
    __declspec(dram_write_reg) long long data;
```

```

    __declspec(dram) long long mem;
    SIGNAL_PAIR sigpr;
    dram_read_write_ind_t ind;
    ind.value = 0;
    ind.ov_byte_mask = 1;
    ind.byte_mask = 0x10; //00010000b;// set up to write lowest order byte.
    mem = 0;
    data = 0x0123456789abcdef;
    dram_write_ind(&data, &mem, 1, ind, sig_done, &sigpr);
    __wait_for_all(&sigpr);
}

```

3.2 Data Allocation

You can declare data with or without allocation attributes. Allocation attributes can describe where the variable is allocated (allocation region) or the scope of the variable. These allocation attributes are provided as `__declspec`s. The allocation region attributes indicate a choice of register or specific memory types where the data needs to be allocated.

For pointers, allocation attributes can be specified both for the pointer itself as well as for the object it points to. This implies that pointers are only compatible if they point to the same allocation region. Pointers declared to types without any allocation region attribute point to SRAM by default.

In the absence of a region allocation attribute variables are allocated to registers in the following circumstances:

- They are 64 bytes (128 bytes in 4-context mode) or less in size, and
- Their address is not taken, or if taken, the address reference is optimized away, (note: array references with non-constant indices implicitly take the address of the array beginning), and
- There are enough registers to accommodate the variables

In the absence of a region allocation attribute, variables are allocated to local memory or SRAM in the following circumstances:

- If there are not enough registers to accommodate all user variables, some are spilled to local memory or SRAM. This includes global variables as well as those local to a function, function arguments, and return values.
- Variables larger than 64 bytes (128 bytes in 4-context mode) are generally allocated to local memory or SRAM.
- An array is allocated in local memory or SRAM, if it uses an index that is computed at run-time. There is no way to index variables in a GPR. The compiler can not use `T_INDEX` to index into xfer registers for various reasons (`T_INDEX` is not per-context, availability, performance, etc.). Similarly addressed variables are allocated to SRAM or local memory if the address reference cannot be optimized away.

The command line option, `-Qperfinfo=2`, provides user-defined variables to register/memory mapping. (See [Table 2, “Supported CLI Option Switches” on page 20.](#))

3.2.1 Register Regions

The following `__declspec()` modifiers can be used to specify allocation to registers:

- `__declspec(gp_reg)` to allocate to a general purpose register
- `__declspec(sram_read_reg)` to allocate to an SRAM read transfer register
- `__declspec(sram_write_reg)` to allocate to an SRAM write transfer register
- `__declspec(sram_read_write_reg)` to allocate to an SRAM read transfer register and to an SRAM write transfer register with the same register number.
- `__declspec(dram_read_reg)` to allocate to a DRAM read transfer register
- `__declspec(dram_write_reg)` to allocate to a DRAM write transfer register
- `__declspec(nn_local_reg)` to allocate to a next neighbor register local to this ME
- `__declspec(nn_remote_reg)` to declare the name of a next neighbor register in the next neighbor Microengine (this is to be used in `-Qnn_mode=0`, i.e., NEIGHBOR mode). The linker patches the physical register number for each reference.

The IXP2XXX NPU cannot allocate a variable to a register under certain situations in which case an error message is produced and compilation aborts. Example reasons for error are:

- If your program takes the address of such a variable
- If the variable is too large (greater than 64 bytes, or 128 bytes in 4-context mode)
- If there are too many variables requiring allocation to registers

In such cases the compiler in addition to reporting this as a user error, reports an indication why it was not able to allocate the variable to a register.

In general, pointers to register objects are not guaranteed to compile successfully, because the compiler needs to be aware of exactly which registers are being accessed at any given time. If the compiler cannot resolve a register pointer access into a “fixed” register access, an error will be generated. The exception is indexed transfer register arrays, described in the “Transfer Registers” section below.

3.2.1.1 General Purpose Registers

The qualifier `gp_reg`, if used in a variable declaration, causes the compiler to allocate general purpose register(s) for that variable.

3.2.1.2 Transfer Registers

The qualifiers for read/write transfer registers, if used in a variable declaration, indicate that you want to associate the variable with specific class of transfer register.

Note: In instructions that normally take a read transfer register, the IXP2XXX has been enhanced to allow both SRAM and DRAM read registers, but only when `-Qnctx_mode=8`.

This enhancement is disallowed when there are only 4 contexts because there are not enough bits in 4 context mode to encode these additional registers.

For example, SRAM memory read operations, which normally take only SRAM read transfer registers when `-Qnctx_mode=4`, can additionally take DRAM read transfer registers when `-Qnctx_mode=8`

This is realized in the compiler with additional intrinsics that contain a `_D` or `_S` suffix. Several examples are shown here.

- `sram_read_D()` - read SRAM using DRAM transfer registers.
- `scratch_read_D()` - read Scratch using DRAM transfer registers.
- `reflect_read_D()` - read reflector using DRAM transfer registers.
- `dram_read_S()` - read DRAM using SRAM transfer registers.

For detailed information on these functions, refer to [Chapter 4, “Intrinsic Functions”](#).

No data can be read from a write transfer register. This implies `char`, `short`, and bit field data types with less than 32 bits in length are illegal to be written into these registers. Note that a write with less than 32 bits requires reading the missing bits from the write register and packing it to a new 32 bits entity before the write.

A variable declared as `__declspec(sram_read_write_reg)` takes both SRAM read transfer register and SRAM write transfer register with the same register number. It is mainly used for inline assembly code where the operand is required to be both an SRAM read and write transfer register. You should be careful to ensure that the value written into that variable goes to the SRAM write transfer register while the value received from that comes from an SRAM read transfer register.

Arrays of transfer registers can be indexed by a variable, as in the following example:

```
__declspec(sram_write_reg) int a[5];
int i;
for (i = 0; i < 5; i++)
{
    a[i] = i;
}
```

This type of access will be compiled into code that uses the `T_INDEX` register to access the transfer register banks.

3.2.1.3 Next Neighbor Registers

The `nn_local_reg` qualifier causes allocation of the variable to a next neighbor register in this ME. If the `NN_MODE` (see option `-Qnn_mode` in [Table 2, “Supported CLI Option Switches” on page 20](#)) is 0 (`NEIGHBOR`), this variable is read-only within this ME program. In this mode, the previous neighbor ME program may declare the same variable with an `nn_remote_reg` qualifier and can only write into it.

When `-Qnn_mode` is 1 (`SELF`) a variable with the `nn_local_reg` qualifier cannot be modified from another ME. (NOTE: there is a 5-cycle latency between a write instruction and a subsequent read instruction from the same NN register with a new value). It can be both read and written within this ME program. In this mode one cannot use the `nn_remote_reg` qualifier.

In the `SELF` mode all next neighbor registers should be declared with `__declspec(nn_local_reg)`. Given this, the following is to be followed. In `SELF_mode`, a `nn_local_reg` variable can be read or written, and a `nn_remote_reg` variable should not be used. In `NEIGHBOR` mode, a `nn_remote_reg` variable can only be written, and a `nn_local_reg` can only be read.

Note: If `-Qnn_mode=0` (neighbor mode) variables declared as `nn_local` or `nn_remote` must be declared in global scope (i.e., they cannot be declared as local variables within a function) because they are referred to from a neighbor microengine.

3.2.1.4 Volatile Registers

The volatile attribute can be applied to variables in transfer registers or to signal variables. This attribute indicates that the register can be read or written by instructions not explicitly referencing the register. An example of this is the status and signal written by the MSF hardware when an incoming packet is sent to a thread. Another example is transfer registers that are read or written by reflect operations from another microengine.

When a register or signal variable is declared volatile, the register or signal that the compiler assigns to that variable will not be used for any other purpose within the scope of that variable. If the variable is global, then its register or signal will be assigned to that variable for the entire program. If it is local to a function, then the register or signal will be assigned to that variable for the scope of that function. This insures that external MEs or hardware will be able to access the variable even though that variable may not be "live," or actively in use, within the currently executing segment of code. If the volatile variable is local to a function, the user will need to provide a synchronization mechanism to insure that the function does not return until the data is no longer needed by the external MEs or hardware. Function-local volatiles may allow the compiler to allocate registers more efficiently than global volatiles, since the register can be reused for other variables once the function returns. In general, volatile variables limit the efficiency of register allocation, and should only be used when necessary. If you wish to indicate that a register or signal variable can be accessed by an external entity only within a certain region of the program, the `__implicit_read()` and `__implicit_write()` intrinsics can be used instead of the volatile attribute.

3.2.2 Memory Regions

The IXP2XXX NPU has six memory regions:

- Local Memory
- SRAM
- DRAM
- Scratch
- R_FIFO
- T_FIFO
- MSF_CTRL

The compiler allows the following `__declspec` modifiers to specify memory to allocate in:

- `__declspec(local_mem)` to allocate to LOCAL MEMORY.
- `__declspec(sram)` to allocate to SRAM
- `__declspec(sramN)` where `n` is 0,1,2 or 3 to specify allocation to a particular SRAM bank
- `__declspec(dram)` or `__declspec(sdram)` to allocate to DRAM
- `__declspec(scratch)` to allocate to Scratch
- `__declspec(rbuf)` or `__declspec(r_fifo)` when declaring a pointer to the RBUF/R_FIFO MSF area

- `__declspec(tbuf)` or `__declspec(t_fifo)` when declaring a pointer to the TBUF/T_FIFO MSF area
- `__declspec(msf_ctrl)` when declaring a pointer to the MSF control area

Examples:

```
__declspec(local_mem) struct msg_header header;
```

declares a variable of type `struct msg_header` which resides in Local Memory

```
__declspec(dram) buffer * buf_ptr;
```

declares a pointer to a buffer data type in DRAM. `buf_ptr` is assigned to a register.

```
__declspec(dram) buffer * __declspec(scratch) buf_ptr_1;
```

declares a pointer to a buffer data type in DRAM. The pointer resides in Scratch.

Note: The memory type modifier applies to the type to its right, thus the first one indicates that the buffer is in DRAM, while the second indicates that the pointer to it is in Scratch. Variables declared in memory using the above syntax can be used with standard C syntax. When the C code reads or writes a variable, the compiler automatically guarantees synchronization. Typically, if the compiler reads a variable, it issues a context swap and waits until the data is available. This swap may be delayed until after other computations that do not depend on the read value to complete. Writes also generate context swaps to prevent read-after-write or write-after-write interference. The compiler may issue multiple writes and reads before a context swap in cases where it can disambiguate the references and guarantee that no conflicts occur.

Note: The `msf_ctrl`, `rbuf/r_fifo`, and `tbuf/t_fifo` regions can only be used to declare pointers; the compiler cannot allocate memory in those regions since the IXP MSF hardware performs this function.

3.2.3 Shared Data

Because the processor supports multiple contexts, you can declare data to be shared between contexts or to be local to each context on a microengine. By default, all variables (both within and outside of a function) are local to a context, thus they are physically duplicated for each of the eight contexts on the ME. In other words, each context has a separate copy of the variable to work with no matter where the variable is allocated (i.e., registers or memory). In addition to this, you sometimes need variables that are shared by all eight contexts on a processor.

The compiler supports this with another `__declspec` modifier:

```
__declspec(shared)
```

The shared attribute can be combined with a memory region attribute in a single `__declspec`, example: `__declspec(shared sram) int x;`

Without a memory region, the shared attribute declares a potential register candidate that is shared by all contexts and is subject to the normal register restrictions.

3.2.4 Global data

You can declare data in SRAM, DRAM, or Scratch memory that is shared by all the microengines on an NPU. One microengine program will "export" (and optionally initialize) the variable with:

```
__declspec(export sram) int i = 42; // initialization is optional
```

The other microengine programs will "import" the variable with:

```
__declspec(import sram) int i; // no initialization possible
```

Although the variable is not "attached" to the microengine that exports it, the export/import qualifiers are needed to prevent multiple initializations of the variable.

3.2.5 Load Time Constants

Load-time constants (i.e. constants that are bound to fixed values at load time and used at run time) are supported through an intrinsic `__LoadTimeConstant(string)`. This is provided as a mechanism for sharing data with the Intel XScale® core. For example:

```
C = a + __LoadTimeConstant("LTC");
```

For each call to `__LoadTimeConstant("LTC")`, the compiler generates a pair of `immed_w0[t, 0]` and `immed_w1[t, 0]` instructions, each with a linker directive on source bits, and uses the temporary variable "t" in the expression (see above example) for the constant "LTC". The linker uses a 32-bit constant for "LTC" to patch those bits (upper 16-bit for `immed_w1` and lower 16-bit for `immed_w0`) when the program is loaded.

This intrinsic is described further in [Chapter 4, "Intrinsic Functions"](#).

3.2.6 Signals

The compiler exposes hardware signal and signal pair registers as special predefined data types, `SIGNAL` and `SIGNAL_PAIR` respectively. Alternatively, you can apply a `__declspec(signal)` or `__declspec(signal_pair)` to a variable of type `int` or a struct comprised of two `ints` respectively. You can declare signal variables and pass them by reference to various intrinsic function calls.

The IXP2XXX NPU microengine provides 15 signals for each execution context. You may have more than 15 signal variables so long as no more than 15 are in use simultaneously in the program. This restriction is imposed because there is no efficient mechanism to temporarily store signals in memory or other registers.

The following example illustrates the use of signal variables.

```
SIGNAL sig;
SIGNAL_PAIR sp;
...
dram_read(dst1, src1, 1, sig_done, &sp);
sram_read(dst, src, 4, sig_done, &sig);
while ( ! signal_test(&sig) )
{
```

```

/* do something*/
}
  ..= dst;
  __wait_for_all(&sp); Section 4.9.1
  ..= dst1;

```

Signals can be shared across functions or across microengines in a limited way using the support provided to determine the signal number allocated to a signal variable or by creating a signal mask. This support is provided through intrinsic functions (`__signal_number()`, and `__signals()` respectively). To associate a signal variable with a specific number, you need to call the `__assign_relative_register` intrinsic. When you use these functions you need to convey additional information to the compiler by using the intrinsics `__implicit_read()` and `__implicit_write()` to indicate the lifetimes of the signals involved. See the description of these intrinsics ([Section 4.9.1](#)) for further details.

3.2.6.1 Signal Variable Restrictions

Signal variables have special properties that make them unlike normal variables. The read access of a signal variable if the signal has been delivered has the potential side effect of clearing this signal. Consequently, there are certain restrictions imposed on the usage of signal variables. These restrictions are listed below.

- These variables cannot be assigned to or used in any way other than as addressed arguments to specialized intrinsic functions or through inline assembly.
- You cannot take the address of a signal variable, except when passing it as an argument to an intrinsic function. However, you can get a signal's address as an int through the `__signal_number()` intrinsic or as a mask through the `__signals()` intrinsic. This can then be passed to a function as an int type parameter. You may also have to insert calls to `__implicit_read()/__implicit_write()` to convey the live ranges of indirectly accessed signal variables, normally in the caller around the call site if there are no other reference to the signal variable. This is because `__implicit_read()` and `__implicit_write()` accept signal variable, and callee function taking such int type parameter usually doesn't know what signal variables that int contains.
- You cannot create an array of signals or `__declspec` any aggregate variable to be a signal. A signal variable if declared explicitly using `__declspec(signal)`, must be declared as a 4 byte quantity such as an int or long. Similarly `__declspec(signal_pair)` can only be applied to an 8 byte data type.
- You cannot read/write, copy, or pass signals as arguments or return values across user function boundaries. Furthermore, as hardware signal registers are context specific, one cannot declare a signal that is shared across contexts. Signal variables can, however, be used for cross-thread communication by declaring them as `__declspec(remote)` or `__declspec(visible)`, or by explicit user allocation using the `__assign_relative_register()` intrinsic.
- A IXP2XXX microengine, when executing the `ctx_arb` instruction with the OR token (`__wait_for_any()` intrinsic), does not clear any of the signals that are asserted. After the use of signal variables in a `__wait_for_any()` intrinsic, you must clear them with calls to `__wait_for_all()` or `signal_test()` intrinsics, prior to reusing the signals.

3.2.7 Local Memory Allocation

3.2.7.1 Overview

You can allocate variables to local memory by applying the `__declspec(local_mem)` type qualifier. These variables may be used in all situations where variables declared in other memory regions may be used, with a few restrictions. First, data in local memory cannot be exported to other microengines. Also, local memory is limited in size: each ME only contains 640 32-bit words of local memory, which must be shared among the running contexts (specified with the `-Qnctx=` or `Qnthreads=` command line options) on that microengine.

In addition to allocating user-qualified variables to local memory, the compiler may also spill (copy) some variables that normally reside in registers to local memory when not enough registers are available for computation. The `-Qperfinfo=1` command-line option will indicate which variables, if any, are being spilled.

3.2.7.2 Placement of Variables

The compiler will make decisions on the allocation and layout of variables declared to be in local memory. The programmer is also allowed to perform manual allocation and access local memory through addresses hard-coded in the program provided this space has been reserved using the `"-Qlm_start=<n>"` command-line option. No assumptions can be made about the layout or relative ordering of individual variables allocated in local memory by the compiler. For example, with the following declaration:

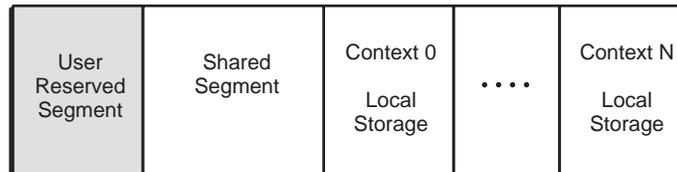
```
__declspec(local_mem) int x, y;
```

You cannot assume that `y` is allocated at the next word following `x`. Any such assumptions should be confined to be between data members of an aggregate (struct/union/array), where the layout of the aggregate is defined by the C language definition.

If one or more instructions reference two variables allocated in local memory, then both those variables can be addressed using the same local memory base register value, provided that the local memory pointer is aligned on a 16-word (i.e. 64 byte) boundary, and the two variables are within the same 16-word aligned block. Therefore, to minimize the cost of instructions involved in setting up a local memory base register, decisions on the placement of variables in local memory relative to one another are optimized based on the usage pattern of the variables. Variables are grouped into "buckets," where all the variables in a bucket are indexed with the same base pointer, to minimize the number of times when a local memory base register has to be reassigned through the `local_csr_write[]` instruction. However, the IXP architecture's OR-based offsetting forces buckets to be aligned on an appropriate word boundary, which can create unused "gaps" between the buckets.

3.2.7.3 Thread Local vs. Shared Storage

Figure 2. Local Memory Layout



B0198-01

The preceding figure illustrates local memory layout. Each "segment" contains "buckets," which are addressed, in the case of thread-local storage, using one base pointer value for each context, and in the case of shared storage, one base pointer value for all contexts. The first segment is optional - you might request that a portion of local memory be excluded from compiler allocation by using the "-Qlm_start=<n>" command-line option. This tells the compiler not to allocate variables to local memory addresses in the range [0, n]. If you do not specify this option, the compiler will start allocating local memory at address 0. The next segment, which the compiler allocates, is the shared segment, which holds all the local memory objects shared by multiple contexts, i.e. the variables specified with the "shared" qualifier. The remaining segments contain the thread-local variables for each context running on the microengine.

3.2.7.4 Viewing Local Memory Usage

The allocation of local memory can be examined with the following compiler option:

`-Qperfinfo=16`

For example, for this program, which allocates an array of 10 integers in thread-local storage and a single integer in shared memory:

```
__declspec(local_mem,shared) int x;

void main() {
    __declspec(local_mem) int a[10];
    ....
    a[9] = x;
    ....
}
```

The allocation information from "-Qperfinfo=16" will be output as follows:

=> User reserved: 0 bytes, Shared segment: 64 bytes, Local page (including gap): 64 bytes

=> Gap between context pages is 24 bytes The data on the page is 40 bytes

Direct access local mem group 0x180d900

Maximum offset used: 36 Alignment: 4

Num members: 1 Total size: 40

[This group contains thread local symbols]

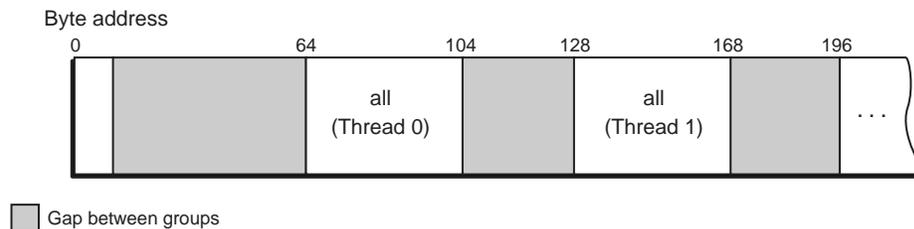
```

lmem.c(8): a allocated at offset 0
Direct access local mem group 0x180d9cc
Maximum offset used: 0 Alignment: 4
Num members: 1 Total size: 4
[This group contains shared symbols]
lmem.c(15): _x allocated at offset 0

```

This information corresponds to the following memory layout:

Figure 3. Local Memory Layout for Program 1



B0246-01

The first part of the allocation information describes the size and placement of each segment in local memory. Assuming that you have not reserved any local memory for manual allocation, the shared segment starts at address 0, and is 64 bytes long. Since there are only four bytes of shared storage (corresponding to the variable `x`) that are actually allocated, the other 60 bytes of the shared segment remain unused. Each thread-local storage segment (called a "local page") is also 64 bytes long. 40 of these bytes are data (the 10-word array) and 24 bytes are the "gap," or unused space.

The next sections of the allocation information describe the object groups (buckets) contained in each segment. As described above, a group contains either shared data, or thread-local data, but not both. If a group contains shared data, all the objects in the group are accessed by all threads using one base pointer, OR'ed with the same offset. If a group contains thread-local data, all the objects in that group are accessed using a different base pointer on each thread, OR'ed with the same offset. So if "group 0" was assigned to thread-local data and contained two objects, `a` and `b`, each thread would contain a base pointer that would point to the beginning of that thread's "group 0," each `a` would be accessed using the same offset for each thread, and each `b` would be accessed using the same offset for each thread (but a different offset than `a`).

The first group in this example contains one thread-local object, the array, `a[]`, declared at source line 8 in the `lmem.c` file. The maximum constant offset used in the group is 36 bytes (corresponding to `a[9]` in the above code). This affects the spacing between groups, as each group must be aligned so that all its indexed elements are addressable with OR-based indexing. The alignment value specified is the C-language-specified alignment of the group's objects; it is 8 bytes for groups containing 64-bit primitive types and 4 bytes for groups containing 32-bit or smaller primitive types. The group is 40 bytes long. The given offset ("0" in this case) of the object `a` is the object's byte offset from the beginning of the entire local page for a given thread, therefore the offset of the first object in a group identifies the alignment of the group.

The next group described is the group containing the shared variable, `x`. The information for this group reads similarly to the other group—the maximum offset used is 0 (since there is only one word of storage used), the alignment is 4 bytes, there is one object in the group, and the group is 4 bytes long. The object "`x`" is allocated at offset 0 from the beginning of the shared data segment.

3.2.7.5 Alignment Information for Local Memory Pointers

In certain cases multiple accesses to local memory objects might be driven by a single `local_csr_wr` instruction. The compiler will initialize base register once and then use different offsets to access different data items allocated in close proximity of that base. To do that, however, the compiler must know the alignment properties of the base address. Without this knowledge, the compiler will not be able to drive multiple pointer accesses through the same base value. Legal values for local memory alignment are 8, 16, 32 and 64. All these values will be interpreted by the compiler as byte addresses. For example:

```
typedef struct
{
    int a;
    int b;
    int c;
    int d;
} MyStruct;

void foo(MyStruct __declspec(local_mem aligned(16)) * P)
{
    ... = P->a
    ... = P->b
    ... = P->c
    ... = P->d
}

main()
{
    MyStruct __declspec(local_mem aligned(16)) X;
    foo (&X );
}
```

In the example given above the compiler might be able to generate just one `local_csr_wr` instruction initializing the base with the value of pointer and then use that base for four different accesses to the fields of the structure. Without having alignment information on pointer `P`, the compiler will have no choice but to drive each single access to a field of the structure through separate base. Therefore, it will generate four `local_csr_wr` instructions and this might result in poor performance. Whenever the same local memory pointer might be used as a base for multiple different accesses, you should declare the pointer with some alignment information.

3.2.7.6 Suggestions for Improving Local Memory Use

To assist the compiler with optimizing local memory usage, you can apply several techniques:

1. Place large objects in the shared segment. Reducing the alignment restrictions on thread-local data will provide space savings for each thread, because all threads have the same layout in their thread-local storage segments.
2. Use `__declspec(align(n))` judiciously as described in [Section 3.2.7.5](#). Larger alignments may cause fragmentation in local memory but it may help the compiler to group several accesses with constant offset (e.g. `p->x`, or `p[3]`) using a single `local_csr_wr[]` to base pointer. Variable addressing (e.g. `p->[i]`, where `i` is a variable) will cause the compiler to do an index calculation

and generate a `local_csr_write[]` instruction plus three nops, which trades off runtime performance for space savings.

3.3 Reflector Inputs/Outputs

A reflector operation involves read/write of transfer registers across MEs and hence across Microengine C programs. Such variables are SRAM/DRAM transfer register variables that are visible across MEs. Hence they are declared with the following declspecs:

```
__declspec(visible sram_read_reg/sram_write_reg/dram_read_reg);
__declspec(remote sram_read_reg/sram_write_reg/dram_read_reg);
```

A Microengine C variable that has a `__declspec(remote sram_read_reg)` refers to a context relative read transfer register in another ME. Note that unlike normal exported variables, which are by definition shared across the contexts of an ME, remote/visible transfer register variables are context relative. Also, remote or visible variables must be declared outside function to avoid name mangling which may confuse the ucl linker when linking.

Signals used for signaling the remote or both MEs are also declared similarly.

```
__declspec(visible) SIGNAL/SIGNAL_PAIR;
__declspec(remote) SIGNAL/SIGNAL_PAIR;
```

When you use `sig_both` to signal both MEs, you must ensure that the same signal number is used for both signals involved. This is done by manually allocating the same signal number to both signal variables involved, using the `__assign_relative_register()` intrinsic as described in [Chapter 4](#):

ME0 Program	ME1 Program
<pre>__declspec(sram_write_reg) me0_x; __declspec(remote sram_read_reg) int me1_x; __declspec(remote) SIGNAL me1_sig; reflect_write(&me0_x, ME1, me1_x, CTX, 1, sig_remote, sig_done, &me1_sig); __free_write_buffer(&me0_x);</pre>	<pre>__declspec(visible sram_read_reg) int me1_x; __declspec(visible) SIGNAL me1_sig; __implicit_write(&me1_sig); __implicit_write(&me1_x); __wait_for_all(&me1_sig); ... =me1_x; //use me1_x</pre>

Notes: A remote read transfer register can only be written in the micro-engine declaring it as remote, and a remote write transfer register can only be read in the micro-engine declaring it as remote.

Calls to `__implicit_write()` need to be inserted in the ME1 program to indicate the earliest point in the ME1 program execution, where ME0 might write the registers associated with visible variables `me1_x` and `me1_sig`.

The receiver ME for a Reflector instruction (the remote ME for Reflector write or the local ME for Reflector read) can use DRAM/SRAM transfer register as reflector operand under 8 context mode, and only SRAM transfer register under 4 context mode; the sender ME always uses SRAM transfer register for reflector instructions.

3.4 Summary of Allowed Data Attribute Combinations

Table 7. Summary of Allowed Combinations of Attributes on Data

	Thread Local	Shared	Export	Remote/Visible
GP Register	Yes	Yes	No	No
Transfer Register	Yes	No	No	Yes
Signal Register	Yes	No	No	Yes
Next Neighbor	Yes	No	No	Yes
Local Memory	Yes	Yes	No	No
SRAM	Yes	Yes	Yes	No
DRAM	Yes	Yes	Yes	No
Scratch	Yes	Yes	Yes	No
RBUF	Yes	No	No	No
TBUF	Yes	No	No	No

3.5 Expressions

In general, all C expression syntax involving the supported data types is supported. Remote XFR register variables can only be used in reflect inline asm and reflect intrinsics. Signal variables can only be used as intrinsic arguments or in inline asm. Function pointers are not supported.

Note: A special note for the implementation of integer divide-by-zero. Because microengines do not support signals nor exceptions, evaluating an expression such as $x/0$ or $x\%0$ for any integer x , signed or unsigned, returns `0xffffffff` for 32-bit integers or `0xffffffffffffffff` for 64-bit integers.

3.6 Statements

The compiler supports all C statements involving supported expressions.

3.7 Functions

3.7.1 Supported

The compiler supports C functions including:

- Local variables with memory regions (equivalent to static locals)

3.7.2 Not Supported

- Recursion

- Variable length argument lists
- Pointers to function
- Passing aggregates larger than 64 bytes (or 128 bytes in 4-context mode) as function arguments or return value

The implementation on recursion and variable length argument lists on the IXP2XXX NPU impacts performance and is therefore not supported. The restriction on function pointers allows the compiler to determine the call-graph exactly and optimize every function call. The use of function pointers requires that all functions that might be called through a pointer have a standard argument passing and return value mechanism. Since aggregates larger than 64 bytes (or 128 bytes in 4-context mode) are never allocated to registers, and function arguments and return values are passed only in registers, the compiler gives an error message on function arguments/return values that are larger than 64 bytes (or 128 bytes in 4-context mode).

3.7.3 Extended Function Attributes

The following function attributes can be applied to functions to define their characteristics with respect to inlining.

- `__noinline func();`
- `__forceinline func(), __inline func();`

The attribute `__noinline` indicates that the function should not be inlined. The attribute `__inline` is a hint to the compiler to inline the program. The attribute `__forceinline` is a strong hint for the compiler to inline the function. Unless optimization or inlining is turned off the function will be inlined by the compiler. `__forceinline` functions are by default static functions. See [Section 2.2.2, “Inlining” on page 19](#) for information on inline `__forceinline` functions.

3.7.4 Optimizing Pointer Arguments

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

```
void foo(MyStruct *p)
{
    some code using *p and assigning *p
}

main()
{
    MyStruct x;
    ...code ...
    foo(&x);
    ...code...
}
```

In the preceding code, you wish 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, which significantly slows access to the data contained in `x`.

One way to write the preceding code, which still allows “`x`” to be placed into registers, is as follows:

```

MyStruct CompilerTemp;

void foo( void )
{
some code directly using and assigning CompilerTemp
}

main()
{
MyStruct x
...code...
CompilerTemp = x;           /* copy -in */
foo();
x = CompilerTemp;          /* copy-out */
}

```

In this code, the program copies x into a global temporary structure that is accessible to both main and foo, allows foo to perform operations on this temporary structure, and copies the results back into x. Both the temporary structure CompilerTemp and the structure x can be placed into registers, with a significant performance gain over the first example.

3.7.4.1 The “restrict” Qualifier

The compiler can automatically perform the structure copy optimization described above if the “restrict” qualifier is applied to the function definition of “foo”:

```

void foo(MyStruct * restrict p)
{
alias-free code using *p and assigning *p
}

main()
{
MyStruct x;
...code ...
foo(&x);
...code...
}

```

The “restrict” qualifier must come directly before the variable name. This qualifier informs the compiler that the memory pointed to by the attached pointer parameter is not accessed through “unknown” means—either through another pointer whose definition is ambiguous, or from another thread. The optimization described above is only guaranteed to be possible, and safe, when the following conditions exist:

- The memory location pointed to by the “restrict” pointer parameter is only accessed using a dereference of that particular pointer, or with copies of that pointer which are defined within the function. The pointer is not assigned to a non-restricted pointer, and the restricted pointer copies, if they exist, are only assigned to once (similar to “const” variables).
- The memory location pointed to by the “restrict” parameter is only accessed from a single thread and a single microengine while the function is executing.
- The “restrict” pointer parameter is not cast to another type of pointer.
- The “restrict” parameter is dereferenced with a constant offset. For example, in the preceding code, the function body can contain *p, p->field, and *(p+2), but not *(p+i) where “i” is not a constant.

The compiler can check for simple violations of the above rules and will not perform the "structure copy optimization" in those cases, but you must determine whether the "restrict" qualifier is appropriate (otherwise the qualifier would not be needed). The option "-Qperfinfo=256" will tell the compiler to print out information on any "restrict rule violations" that it finds.

3.8 User Assisted Live Range Analysis

Register allocation and other compiler optimizations depend on correct live range information of register variables to make the right decision. A register variable is defined as a variable that could be assigned to a register (transfer register, signal register, general purpose register). 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. Each live range covers one definition and its sequential reads.

The live range of a register variable begins with a write into the variable; and it terminates at the point where there is no subsequent read of that 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. It could have different physical registers assigned to it across different live ranges. You cannot have another write into the same variable in the middle of a live range (otherwise the live range would be split), but you could have multiple reads in the middle of a live range. Once past the last read, the live range is terminated and the physical register can be released.

A register variable can have a write without a read, or a read without a write. For example:

Example A	Example B	Example C
<pre>main() { __declspec(sram_write_reg) x; SIGNAL s1; // read x without // write to x first sram_write(&x, &p, 1, ctx_swap, &s1); };</pre>	<pre>main() { __declspec(sram_read_reg) x; SIGNAL s1; // write into x // no read of x sram_read(&x, &p, 1, ctx_swap, &s1); }</pre>	<pre>main() { __declspec(sram_read_reg) x; SIGNAL s1; // write into x sram_read(&x, &p, 1, ctx_swap, &s1); // write into x again sram_read(&x, &q, 1, ctx_swap, &s1); // read x ... = x }</pre>

In example A, x is read without being written, so the compiler assumes the live range starts from the beginning of the program. In example B, x is written but not read; the compiler concludes that the write is redundant and can be removed during optimization. Example C is the variation of example B. X is rewritten before the value from the first `sram_read` is used. The compiler can remove the first `sram_read` as a redundant instruction. Even if it is not removed for other reason, the compiler will release the physical register x immediately after this instruction.

In summary, a write always terminates an old live range, and may start a new live range if there is a preceding read. And, a read always extends the live range of a register at least to the read point.

One of the challenges this process brings to the compiler is the implicit read/write of a register. Normally, you can only modify or reference a register through explicitly expressed names, like register `x` in our previous examples. This process provides you with ways to implicitly read and write a register without referring to the register name. The scenarios include but are not limited to the following:

- A signal/Xfer register is defined on a remote ME and used on local ME. The definition/write is not visible from the local program.
- A signal/Xfer register is defined locally and used on a remote ME. The reference/read is not visible from the local program.
- A signal/Xfer register is assigned an absolute register number. It could be read/written without referring to the symbolic name.
- A signal has the signal number exposed through `signals()` or `signal_number()`. It could be read/written without referring to the symbolic name, for example, through `local_csr` write.
- A xfer register has address taken and used in indexing reference through `T_INDEX` (not supported in PR3).
- A NN register that being referenced indirectly through NN register ring.
- A synchronized I/O operation with `sig_done`. For example:

```
__asm sram_write[x, &p, 0, 2], sig_done[s1];
...
__asm ctx_arb[s1];
```

Even if this instruction is the last use of the xfer register(s), you should not release the xfer register(s) immediately. The semantics of I/O instructions requires you to hold the xfer register(s) until the signal arrives. So in our example, the live range of `x` needs to be extended to pass `ctx_arb`.

Under certain scenarios the compiler cannot not detect the correct live range of a register. For example, if the compiler prematurely terminates the live range of a register, it could overwrite the value currently in use; or if the compiler prolongs the live range of a register, it could run out of register space unnecessarily. You must supply live range directives for the compiler to make the right decision.

There are three compiler directives for liveness computation:

- `__implicit_read()` will prolong the live range to at least the point where `__implicit_read()` is called. If there are other reads of the same register after this point, then adding `__implicit_read()` does not have any effect on the program.
- `__implicit_write()` will terminate a live range, and start a new one if there are following reads. If `__implicit_write()` is followed by another write, and there is no read in between, then this `__implicit_write()` has no effect on the program.
- `__free_write_buffer()` will free the I/O buffer (xfer register) only if there is no other read of it after this point. See [Section 7.2, “Things to Remember When Writing Microengine C Code” on page 374](#) for examples of the use of `__free_write_buffer()` and `__implicit_read()`.

Note: For syntax, please see the intrinsics in [Section 4](#).

3.9 Viewing Live Ranges

The command line option `-Qliveinfo` displays live-range information about a set of register classes. You can use `-Qliveinfo` or `-Qliveinfo=all` to display live-range information for all register classes, or use `-Qperfinfo=<reg_class,...>` to display live-range information for a selected set of register class. [Section 2.3.3, “Supported Compiler Option Switches” on page 20](#), explains the syntax of the option in detail.

If a variable is live at some point in a program, this means that the variable's value is used somewhere after that point in the program, and therefore storage (registers in this case) must be reserved for that variable. If too many variables are “live” at the same point in a program, not all of them will be able to be stored in registers, and some of them will have to be “spilled,” or demoted, into local memory, NN registers, or SRAM. This can adversely affect performance. When this happens, `-Qliveinfo` can help you analyze your program and determine which code segments have a high “register pressure” and thus need to be restructured.

Every object in your program that is assignable to registers, including user variables, is represented in a common format, the “virtual register.” Virtual registers have a class, ID number, and optionally a user variable name. Register allocation is the process of mapping virtual registers to physical ones. The `-Qliveinfo` printout provides information on these virtual registers in the following format:

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

where “cls” is one of the register class names specified in the `-Qliveinfo` parameter set, “ID” is the ID of the VR, and “variable_name” is the name of the corresponding user variable, if any.

`-Qliveinfo` prints the live-range info for each register class separately. For a given register class, the first part of its live-range display is a mapping from the virtual registers of that class to the corresponding user variable names. Not all virtual registers can be mapped to a variable name. For those that cannot be mapped, an ellipsis (...) is used instead. Some VRs will correspond to compiler-generated variables (usually of the form “cgt.nnn”). Global variables in your code will have their names modified slightly.

The following is an example of the virtual register map:

```
Virtual gpr Registers to User Variables Mapping:
nn_inl_01a.c(30)  _x      gr.123
nn_inl_01a.c(32)  _y      gr.328
nn_inl_01a.c(32)  _y+4    gr.329
nn_inl_01a.c(32)  ...     gr.330
```

After the virtual register map, the live-range info is printed for each pseudo-assembly instruction in each function. At the top of each function display, three sets of VRs are printed: the live-in set, the live-out set, and the live-through set. The “live-in” set is the set of VRs that are live at the points where the function is called. “Live-out” is the set of VRs that are either “live-in” or defined in the function, and are used after the function returns. “Live-through” is the set of VRs that are “live-in” and “live-out,” but not referenced inside the function. “Live-through” reflects the register pressure from the function's callers. The function-level liveness information is then followed by the live-

range information for each pseudo-assembly instruction (pseudo-assembly instructions are used internally by the compiler, and correspond roughly with IXP assembly instructions, but may have different syntax).

An example of the live-range info for a function is:

```
Live info. of gpr registers for Function test1b:
  Live in(1):
    gr.671(..)

  Live out(0):

  Live through(0):

  Live set(1): gr.671(..)

  /*****/      puthi(S1); put('\n');
1      immed[gr.348(val) , 65535, <<0]
  Live set(2): gr.348(val) gr.671(..)

2      immed_wl[gr.348(val) , 32767]
  Live set(2): gr.348(val) gr.671(..)

3      .mcall[_puthi#, gr.663(..) ]
  Live set(1): gr.671(..)

4      immed[gr.328(c) , 10, <<0]

  Live set(2): gr.328(c) gr.671(..)

5      .mcall[_put#, gr.655(..) ]
  Live set(1): gr.671(..)

  /*****/      puthi(S1/(1<<0)); put('\n');
6      immed[gr.348(val) , 65535, <<0]
  Live set(2): gr.348(val) gr.671(..)
```

Each pseudo-assembly instruction is preceded by a set of the VRs that are live at the point before the instruction executes. The numbers that occur after the set names ("Live set(n)") indicate the number of relative registers needed to allocate the live VRs at that point. "Shared" VRs or variables will count as a fraction of a register, because several absolute registers can be mapped into a single relative register. In the above example, the VR gr.348 is live between instructions 1 and 3, and live after instruction 6. Instruction 1 writes to the VR, which makes it live afterward (recall that liveness is an indication of whether a VR's value is needed). Instruction 3 is a function call to puthi(), which uses the value of gr.348 (this fact would be apparent on examination of the live-range info of the puthi() function). After the function call, gr.348 is no longer live because its value was only needed for the function call. gr.348 becomes live again after instruction 6, because the VR is rewritten with different value for the next puthi() call (which is not shown).

3.9.1 Limitations and Restrictions on Viewing Live Ranges

- Pseudo instructions do not exactly reflect the assembly instructions in the final list file. Register allocation, local memory allocation, scheduling, and other optimizations might delete, add, modify, or reorder some instructions. Some of the pseudo instructions are compiler directives only, and will not produce any physical instructions in the final list file.
- At a given program point, register pressure slightly greater than the number of available registers does not necessarily mean that a spill (demotion to another storage class) will be generated for one of the live variables. Some variables may be found to be equivalent to each

other and may share the same register. Conversely, register pressure which is slightly smaller than the number of available registers does not guarantee that no spilling is needed, because additional registers may need to be allocated to perform operations such as memory I/O, or to temporarily hold values from other register classes.

- Not all virtual registers can be mapped back to user variables.
- Some variables have modified names that cannot be found in the original source code. They are either global variables or compiler-generated variables.
- When the compiler allocates registers to variables, transfer registers are allocated first. If there are not enough transfer registers to hold all the live transfer register values at a given program point, some of the values will be stored (spilled) in GPRs instead. This may in turn cause spills (demotion to NN registers or memory) in the GPRs that are live at that point. Since the `-Qliveinfo` output is printed before register allocation, it might show a GPR register pressure which is smaller than the number of available registers at a given program point, even though a spill is generated at that point. If this happens, the register pressure of the transfer registers should be examined. Any transfer register pressure larger than the number of physical registers should be added into the GPR register pressure.
- If your code contains local memory variables, 2 GPRs are reserved from the allocation pool, so that local memory address calculations can be performed.
- Variables declared as volatile are marked as live at every point within their scope. Function-local volatile variables are live within the scope of their defining function, and global volatile variables are live within the scope of the entire program.

3.10 Critical Path Annotation and Code Layout

On the IXP architecture, there is a penalty paid for each branch that is taken, i.e. each time the code does not proceed sequentially. This can sometimes be removed by use of branch defer slots, but the compiler is not always able to completely fill the defer slots.

Code layout is an optimization performed by the compiler that arranges the code in an order that reduces the number of taken branches. As an example, look at the following code:

```
if (condition)
{
    <statement 1>;
}
else
{
    <statement 2>;
}
```

Typically, the compiler would produce code similar to this:

```
alu [--, --, b, condition]
bne [lab1#]
<statement_1>
br [lab#]
lab1#:
<statement 2>
lab2#:
```

Notice that there is a taken branch on each path.

Now let's assume that the compiler knows that the 'else' clause is executed far more frequently than the 'then' clause. The compiler could arrange the code differently as follows:

```
alu [--, --, b, condition]
beq [lab1#]
<statement 2>
lab2#:
...
lab1#:
<statement 1>
br [lab2#]
```

Now there are no branches when the 'else' clause is executed and 2 branches when the 'then' clause is executed. This is a win on the average since the 'else' clause is much more frequently executed. In fact, this optimization will always win when the 'else' clause is taken 2/3 of the time or greater.

Another case is the switch statement. A switch is fairly expensive to implement because it involves an indexed branch to a branch. In the case of a switch, if one leg of the switch is taken more than 30% of the time, a test for that leg before doing the switch will improve the code.

The compiler cannot do this optimization without direction from the programmer. This is because these transformations would hurt performance if the execution ratios were not what the compiler assumed. For this reason, the compiler provides an intrinsic to mark the “critical path” in the program, that is the path that is executed most frequently.

Use the intrinsic function:

```
__critical_path()
```

to indicate that this point in the program is on the critical path. For code that is on the critical path, you should mark the leg of a two way branch (e.g. if statement) that is taken 2/3 of the time or more, and for a switch statement you should mark the most important leg if it is taken 1/3 of the time or more. You should not mark any two paths that are mutually exclusive as both critical, as this provides no significant information to the compiler

The compiler is capable of inferring, from the `__critical_path()` directives that you insert, other parts of the program that are on the critical path. For example, if you have a series of nested ifs you only need to mark the leg of the innermost one as being on the critical path. If one critical path will overlap another one, the user may want to set priorities on the paths. Please see [Section 3.10.1, “Multiple Critical Paths”](#) on page 61 for more details.

Here are some rules you should take into consideration when marking the critical path:

- Put a critical path marker inside the main loop of the program, at the top.
- For an if with an else, mark one side or the other if it is executed 2/3 of the time or more.
- For a switch, mark the most often taken case if it is taken 1/3 of the time or more.
- If you have an if without an else, put a critical path marker in the 'then' clause if the if is taken 2/3 of the time or more
- If you have an if that ends with a return or goto statement, and the if (and hence the return or goto) is not executed 2/3 of the time or more, mark the statement following the statement or block controlled by the if.

- If you have a function that is used both on the critical path and off the critical path, do not put critical path markers in the function.

Basically, you want to walk through the code for your program following the most frequently taken path, and place a marker whenever the code makes a decision and one path is on the critical path and the others are not.

3.10.1 Multiple Critical Paths

If several critical paths overlap each other, the branches on the overlapping sections will be laid out in an arbitrary order. For example:

```
if (cond1) {
    __critical_path();
    // block 1: most frequent case
}
else {
    if (cond2) {
        __critical_path();
        // block 2: second most frequent case
    }
    else {
        // block 3: infrequent case
    }
}
```

In the above segment of code, the user wants the "if (cond1)" statement to give preference to "block 1". The user also wants the "if (cond2)" statement to give preference to "block 2". If the `__critical_path()` directive is used as above, the critical path choice at "if (cond2)" will be extended upwards by the compiler and overlap with the critical path choice at "if (cond1)". The compiler will not know how to choose the default branch direction for "if (cond1)". This is by design; if the critical paths were not extended in this fashion the user would have to insert a directive inside every if statement surrounding a given frequently executed block.

When critical paths overlap, the user can tell the compiler which one to give preference to by assigning a priority to each path. The `__critical_path()` directive takes an optional integer argument, which specifies the priority of that path. For example:

```
if (cond1) {
    __critical_path(20);
    // block 1: most frequent case
}
else {
    if (cond2) {
        __critical_path(1);
        // block 2: second most frequent case
    }
    else {
        // block 3: infrequent case
    }
}
```

The numbers can range from 0 to 100. The default is 100 if no argument is specified. The critical path with the higher number is given priority. In the above example, "block 1" will be placed as the default for "if (cond1)" because it has a higher priority (20) than the critical path that flows through "if (cond2)".

3.11 User-Guided switch() Statement Optimization

You can supply information that will determine how the compiler will perform certain optimizations. Among these are default case removal and switch block packing.

Given the following code:

```
void main()
{
    __declspec(sram) int mem[10] = {0,1,2,3,4,5,6,7,8,9};
    int x = 0;
    switch (mem[0])
    {
        case 0:
            x = 1;
            break;
        case 1:
            x = 2;
            break;
        case 2:
            x = 3;
            break;
    }
}
```

The compiler will generate code such as the following:

```
sram[read, $0, a0, 40, 1], ctx_swap[s1]
alu[--, 2, -, $0]
blo[l_10#], defer[1]
alu[a0, --, B, $0]
jump[a0, l_21#, targets[l_23#,l_22#,l_21#]
l_21#:
    br[l_4#]
l_22#:
    br[l_6#]
l_23#:
    br[l_8#]
l_4#:
    br[l_10#], defer[1]
    immed[a2, 1, <<0]
l_6#:
    br[l_10#], defer[1]
    immed[a2, 2, <<0]
l_8#:
    immed[a2, 3, <<0]
l_10#:
    ....
```

The preceding code example shows two possible optimizations that the compiler can perform:

1. The code to test and handle the case where the switch() value does not match any of the other specified cases is not needed.
2. Instead of having the code `jump[]` to a jump table which then branches to the code to handle each case, the `jump[]` can go directly to the handler code, at an offset based on the value of `x`. This optimization can be performed because each case is handled by code that is approximately of equal length (two instructions). Therefore the offset for each handler is equal to the value of `x` times two.

The compiler will perform the above optimizations based on the input you supply.

3.11.1 Default Case Removal

The first optimization in this example, removal of the handler code for the unmatched (“default”) case, requires that you select and provide the appropriate value of the switch() argument. You direct the compiler to remove the “default” case by creating an empty default case and annotating it with the intrinsic function `__impossible_path()`. For example:

```
void main()
{
    __declspec(sram) int mem[10] = {0,1,2,3,4,5,6,7,8,9};
    int x = 0;
    switch (mem[0])
    {
        case 0:
            x = 1;
            break;
        case 1:
            x = 2;
            break;
        case 2:
            x = 3;
            break;
        default:
            __impossible_path();// add default case, and annotate with intrinsic.
    }
}
```

3.11.2 Switch Block Packing

The second optimization in this example (switch block packing) should only be performed if all case handlers are approximately equal in length, with that length preferably a power of two. If the `__switch_pack()` intrinsic function is placed in the default case, the compiler will try to predict whether the code will benefit from switch block packing, and will perform the optimization if this is possible. For example:

```
void main()
{
    __declspec(sram) int mem[10] = {0,1,2,3,4,5,6,7,8,9};
    int x = 0;
    switch (mem[0])
    {
        case 0:
            x = 1;
            break;
        case 1:
            x = 2;
            break;
        case 2:
            x = 3;
            break;
        default:
            __switch_pack(swpack_auto);
    }
}
```

The possible arguments for the `__switch_pack()` function are described in the `swpack_t` enum in the `ixp.h` header file:

```
typedef enum {
    swpack_none, // no pack, jump[] to a jump table
    swpack_lmem, // no pack, but use local memory to hold jump table
    swpack_auto, // auto pack when appropriate
    swpack_0,    // pack if no extra registers are required
                // to perform the jump[] offset calculation
    swpack_1,    // pack if at most 1 register is required to
                // perform the jump[] offset calculation
    swpack_2,    // pack if at most 2 registers are required
                // to perform the jump[] offset calculation
    swpack_3     // pack if at most 3 registers are required
                // to perform the jump[] offset calculation
} swpack_t;
```

Note that this optimization should not be performed if the case handlers vary widely in length, because the smaller handlers will have to be padded so that all handler offsets occur at the same intervals.

3.12 Creating Context Swap-Free Regions of Code

The `__no_swap_begin()` and `__no_swap_end()` intrinsics can be used to create a section of code where the compiler will not create any instructions that incur a context swap, or move any code into the region that will incur a context swap. This allows the user to write critical sections without incurring the overhead of explicit synchronization. Note that the other microengines on the NPU will still continue to execute in parallel. To create a context swap-free region, simply place the `__no_swap_begin()` and `__no_swap_end()` intrinsics at the beginning and the end of the desired section of code, as shown in the following example:

```
__no_swap_begin()
... critical section code ....
__no_swap_end()
```

If the code within the critical section contains a context swap operation, the compiler will generate an error message. This includes any access to data structures stored in memory. Function calls made in the critical section are also checked for this condition. Aside from this checking, the compiler will also guarantee that no other code that incurs context swaps will be moved into this region through compiler optimizations.

3.13 Loop unrolling control

When the `-O2` (compile for maximum code speed) option is enabled, the compiler can perform an optimization called "loop unrolling" as shown in the following example.

Original loop:

```
for (i = 0; i < 10; i++) {  
    a[i] = i;  
}
```

Loop unrolled by 2X:

```
for (i = 0; i < 10; i += 2) { // unroll by 2  
    a[i] = i;  
    a[i+1] = i+1;  
}
```

The loop in the second code segment has been unrolled by 2X (the "unroll factor" is 2). Two iterations of the original loop will execute in one iteration of the unrolled loop. The total number of branches executed in the loop is halved. Also, the two statements in the loop body can be optimized together—computation can be reused and more scheduling and pipelining opportunities have been created. Loop unrolling therefore improves the performance of loops, at a cost in code size.

Loop unrolling is performed only for "for" loops. If loops are nested, only the innermost loop will be unrolled. The compiler automatically determines, using various heuristics, whether a benefit can be had for unrolling a given loop, and what the proper "unroll factor" should be. If you want more precisely controlled unrolling behavior, there are two `#pragma` directives that you can use, as shown in the following example::

```
#pragma nounroll// Don't unroll this loop  
#pragma unroll (<unroll factor>)// Unroll this loop by the given unroll factor
```

These directives are placed directly before the loop to be managed. This loop must be a "for" loop, and must be the innermost loop in a series of nested loops. This is shown in the following examples:

```
#pragma nounroll
for (i = 0; i < 10; i ++) // Don't unroll this
    ...

#pragma unroll(2)
for (i = 0; i < 10; i ++) // Unroll this loop by 2X, exactly as in
// the above example
    ...

                                // NOT LEGAL, must be applied to
#pragma unroll(2)                // innermost loop
for (i = 0; i < 10; i++)
    for (j = 0; j < 6; j++)
        ...
    ...

#pragma unroll(2)// NOT LEGAL, must be applied to for loop
while (1)
    ...
```

The "unroll factor" parameter is the total number of iterations of the loop body that will be in the final unrolled loop. If this parameter is 0 or 1, no unrolling will be performed.

3.14 Mixing C and Microcode in One Microengine

3.14.1 Command Line Options and Usage model

A typical application that mixes C and microcode defines one or more functions in C files, and one or more microcode blocks in assembly file. A main() function must be defined in C. Global variables require initialization and must be defined in C. Thread local memory variables must be defined in C files as there is no thread local memory in microcode. A function in microcode is a label that you can jump to. C functions call microcode functions by first setting arguments and returning address in symbolic registers, then jumping to the label. Microcode can also call C functions in a similar way. The returned value is placed in symbolic registers as well.

Any file scope global variables defined in C can be referenced from microcode. Any microcode module level global register variable, or module level memory variable can be referenced from C functions. The naming translation scheme between C and microcode is described in [Section 3.14.2](#).

The following command line option supports mixing C and microcode in one compilation:

```
-uc
```

With this option on, you can pass a number of C files, together with a number of microcode files to the compiler. First, all C files are compiled into a single microcode file, usually named as ipo_out.uc, and a header file, usually named as ipo_out.h. This header file contains macro definitions of offsets for thread local variables. The compiler then creates a temporary, temp.uc, as the high-level microcode file. Temp.uc includes ipo_out.uc, ipo_out.h, (unless they are renamed by -Fa or -o) and all other microcode files passed on the command line. Compiler later invokes UCA (microcode assembler) to assemble temp.uc into a list file.

After compilation, temp.uc is removed, but ipo_out.uc and ipo_out.h is preserved in the directory.

The following command line options can change the default behavior of mixing compilation:

-S	The driver will not invoke UCA, only generate the uc file.
-Fa is renamed to <filename>	<filename> or -o <filename> The microcode file produced by compiler
-O, -O2, -O1	The driver will invoke UCA with -O
-Zi	The driver will invoke UCA with -g
-Gx2800/-Gx2400	The driver will invoke uca with -ixp2800/-ixp2400
-Fe<filename>	The list file produced from UCA is renamed to <filename>.

3.14.2 Naming and Calling Conventions

To avoid name conflicts with microcode, the compiler inserts a leading underscore ('_') to all C variables and functions. For example, a function foo() in C source file is referred as _foo# in assembly; a global memory variable x is referred as _x in assembly. Additional rules regarding naming convention on variables are explained in following subsections.

3.14.2.1 Register Variable Naming Conventions

When referencing register variables across the boundary of C and microcode, the following conventions are to be followed:

- For SRAM transfer registers, the microcode requires a prefix name of "\$". For the same SRAM xfer variable myvar in C, the corresponding name is \$myvar in microcode.
- For DRAM transfer registers, the microcode requires a prefix name of "\$\$". For the same DRAM xfer variable myvar in C, the corresponding name is \$\$myvar in microcode.
- For next neighbor registers, the microcode requires a prefix name of "n\$". For the same nn reg variable myvar in C, the corresponding name is n\$myvar in microcode.
- For shared registers, the microcode requires a prefix name of "@". For example, the following C variable:

```
__declspec(shared gp_reg) rr;
```

is translated into microcode as shown:

```
.reg read @_rr;
```

In C, register variables can carry a size bigger than a 4 byte word; while in microcode, the size of each register variable is 4 bytes. When referencing such a variable from microcode, a postfix of "_<n>" where n=0,1,2,..., is appended to distinguish each 4 byte part. For example, the following C variable

```
__declspec(sram gp_reg) rr[4];
```

is translated into microcode:

```
.reg read $_rr__0
```

```
.reg read $_rr_1
.reg read $_rr_2
.reg read $_rr_3
```

3.14.2.2 Sharing Variables Between C and Assembly

All register variables may be shared between the C and assembly source files in the program. All memory variables declared with the declspec qualifiers “shared”, “import”, or “export” may also be shared between the C and assembly source files. In the assembly file, all variables being shared should be prefixed with an underscore “_”. The variables should only be declared once per program. For example:

.c File:

```
...
    __declspec(sram, shared) int i;
    __declspec(sram_write_reg) int reg;
...
```

.uc File:

```
; no declarations for i or reg
...
    immed[$_reg, 12]; reg is an xfer register
    sram[write, $_reg, _i, 0, 1], ctx_swap[somesig]
...
```

3.14.2.3 Calling Conventions

Due to lack of stack, parameter passing, return address passing and returning value passing for global functions are via global variables. For example, the call to `x = foo(y)` will be translated into:

```
    alu [ _foo_arg_0, --, B, _y]
    load_addr [ _foo_raddr, L_123#]
    br[_foo#]
L_123#:
    alu [ gr300, --, B, _foo_ret_0]
    ... ..

_#foo#:
    alu [ gr100, --, B, _foo_arg_0]
    ... ..
    alu [ _foo_ret_0, --, B, gr200]
    rtn [ _foo_raddr]
```

`<func_name>_arg_<n>`, `<func_name>_raddr`, `<func_name>_ret_<n>`, where `<n>` is 0,1,2,3... are compiler generated global variables for passing arguments, return address and return value. Each of these is a 32-bit register. You must set these variables correctly when calling a C function from microcode; and you must get the correct arguments and return address when writing a microcode function callable from C.

It is possible for C to declare function arguments as transfer registers explicitly or implicitly. In that case, the corresponding microcode name must have "\$" or "\$\$" inserted to the name begin. The implicitly declaration of argument variable as transfer register happens with inline assembly. For example,

```

Foo(int x)
{
    ...
    __asm sram[write, x, addr1, 0, 1], ctx_swap[s1];
    ...
}

```

The compiler will read x as declared as __declspec(sram_write_reg)

For a global function, saying foo(), value passing symbolic variables, such as _foo_arg_0, _foo_raddr, _foo_ret_0, are defined in the module where the function body is defined. For other modules that did not see the definition of foo(), but has calls to foo(), these value passing variables are declared as extern.

This calling convention is not need for C static functions, because they cannot be called from the assembler.

3.14.3 Mixed C and Microcode Examples

3.14.3.1 Function Parameter Passing

The C file calls an assembly function performing addition of two operands.

```

int x, y, z;
extern int sum(int a, int b);
main()
{
    x = 10;
    y = 5;
    z = sum(x, y);
}

```

The assembly file:

```

;Function _sum: perform add on two argument and return
;the result
.num_contexts 4
.reg global _sum_ret_0
.reg global _sum_raddr
.reg global _sum_arg_1
.reg global _sum_arg_0
.begin
.reg _gpr68
_sum#:
    alu[_gpr68, _sum_arg_0, +, _sum_arg_1]
    alu[_sum_ret_0, --, B, _gpr68]
    rtn[_sum_raddr]
.end

```

3.14.3.2 Register Usage

The C file defines an SRAM transfer register variable.

```
__declspec(sram_read_reg) x[4];
```

The microcode function that returns the sum of all elements of x:

```
.reg extern read $x_0
.reg extern read $x_1
.reg extern read $x_2
.reg extern read $x_3
.reg global _sum_ret_0
.reg global _sum_raddr

; function sum returns the sum of all elements
; of x
_sum#:
.begin
    alu[_sum_ret_0, --, B, $x_0]
    alu[_sum_ret_0, _sum_ret_0, +, $x_1]
    alu[_sum_ret_0, _sum_ret_0, +, $x_2]
    alu[_sum_ret_0, _sum_ret_0, +, $x_3]
    rtn[_sum_raddr]
.end
```

3.14.4 Restrictions on Mixing C and Microcode

There are several restrictions in mixed C/microcode programming. These restrictions include:

- The C portion must contain main()
- You must give all C files to the compiler at compilation time. Otherwise, global variables and thread local storage pointer may not be initialized correctly.
- UCA declares and uses paired signals implicitly. If a signal appears in a position where a paired signal is required, then it is a paired signal and will be allocated two physical signals. There is no way to reference only the odd/even part of a paired signal. For C programs that explicitly reference the odd/even part of a paired signal, the behavior is undefined.
- The calling convention implemented for mixing C/microcode is not compatible with the implementation in inline assembly. This means inline assembly has to follow this convention to call a function defined in microcode.
- If you change nn-mode in assembly, which is not consistent with the C compiler option, the result is undefined.
- UCA does not support case sensitive variable/function name yet. All C symbols are translated into lowercase when referenced from microcode. When local memory is referenced in both C files and microcode files, you should use -Qlm_start to partition local memory between the microcode and compiler code to avoid conflicts.

3.15 Unsupported ANSI C99 Features

The Microengine C compiler is largely based on the ISO/IEC 9899:1990 (C89) standard, with selected features from ISO/IEC 9899:1999 (C99). The majority of the new features in C99 are not supported in the current implementation of the Microengine C compiler, which include the following:

- variable-length arrays
- flexible array members
- static and type qualifiers in parameter array
- complex and imaginary data types
- compound literals
- designated initializers
- preprocessor arithmetic done in `intmax_t/uintmax_t`
- mixed declarations and code
- new integer constant type rules and integer promotion rules
- macros with a variable number of arguments
- trailing comma allowed in enum declaration
- inline functions (Microengine C compiler does implement `__inline`, which is similar to the inline keyword defined in C99)
- boolean type (Microengine C compiler defines `bool` as `int`, which is not the same as `_Bool` defined in C99)
- idempotent qualifiers
- empty macro arguments
- new struct type compatibility rules
- `_Pragma` preprocessing operator
- `__func__` predefined identifier
- `VA_COPY` macro
- LIA compatibility annex
- conversion of an array to pointer not limited to lvalues
- relaxed constraints on aggregate and union initializations
- flag error on return without expression in functions return value (and vice versa)
- no implicit function declaration

In addition to these unsupported features, the Microengine C compiler has the following restrictions:

- It does not support any floating type, including `float`, `double`, and `long double`.
- It does not support recursive function calls.
- It does not support variable function arguments.
- It does not support `setjmp/longjmp`.

- It does not support any signal() functions.
- It does not support taking function address and using function pointers
- Due to the size of control store, it does not support translation limit (such as the number of identifiers, number of nested blocks, etc.) defined in C89 and C99.
- An array subscription on register variables (gp_reg, nn_reg or xfer_reg) can only take a constant value.
- The address operator (&) can not be applied to register variables (gp_reg, nn_reg or xfer_reg), except as an argument to intrinsic function calls.
- Read xfer register variables can not be used as an Lvalue except in a call to intrinsic functions.
- Write xfer register variables can not be evaluated (used as Rvalue) except in inline assembly.
- Most standard library headers defined in C89/C99 are not supported at this time. Only <memory.h>, <string.h> and <stdlib.h> are implemented with significantly fewer functions supported than required by C89/C99.
- A limited number of standard library functions are implemented in the Microengine C compiler. Please refer to <memory.h>, <string.h>, and <stdlib.h> in the *Microengine C Compiler LibC Library Reference Manual* for information on which functions are implemented.

Intrinsic Functions

4

Intrinsic functions support features of the Intel® IXP2XXX Network Processors that are not easily accessible using standard C. They are either expanded in-line in the compiler or defined and inlined in intrinsics. Also defined are many enumerated data types and structures to encapsulate signal and CSR names or other such special information.

The header `ixp.h` defines the C compiler intrinsic functions. The descriptions here are not intended to be complete descriptions of the intrinsic functions—they must be supplemented with the information in the *IXP2400/IXP2800 Network Processor Family Microcode Programmer's Reference Manual*.

There are restrictions placed on the data argument specified to CSR and intrinsic functions. These restrictions are discussed throughout this chapter and in [Section 4.10, “Restrictions On Intrinsics” on page 354](#).

The intrinsics fall into the following categories:

- Unaligned data access
- Memory and I/O access
- Synchronization
- Local CSR, CAM, CRC access
- Miscellaneous functions

The intrinsic functions are known to the compiler and do not generate a function call. They generally translate to one or two machine instructions, excluding evaluation of the arguments.

4.1 Intrinsic Syntax Conventions

Many intrinsics described in this chapter can work with data in either SRAM or DRAM transfer registers. These intrinsics have variant intrinsic names that are denoted with `_D` (for DRAM registers) or `_S` (for SRAM registers). For example,

```
sram_read()  
sram_read_D()
```

These intrinsics perform the same operation except that `sram_read()` reads data from an address in SRAM into an SRAM read transfer register while `sram_read_D()` reads data from SRAM into a DRAM read transfer register.

In this chapter, the syntax descriptions of these variants are combined in one description and are shown as follows:

```
void sram_read[_D] (  
    __declspec([sram, dram]_read_reg) void * data,  
    volatile void __declspec(sram) * address,  
    unsigned int count,  
    sync_t sync,  
    SIGNAL* sig_ptr);
```

The brackets denote the variant syntax. If you use the `_D` variant above, you must also specify a DRAM read transfer register for the data argument (i.e., `__declspec(dram_read_reg) void *data`).

If you use the `sram_read()` intrinsic, you must specify an SRAM read register for the data argument (i.e., `__declspec(sram_read_reg) void * data`).

4.2 Unaligned Data Access

The functions described in the following sections allow access to data that is not aligned on natural boundaries (32 or 64 bits). They also allow access to data types smaller than those supported by the hardware (8, 16, 32 and 64 bits) at any byte address.

These functions take a two-part address:

- A pointer, aligned or unaligned
- An integer byte offset from the pointer.

For functions that do not specify a particular memory region in its name or argument list, the pointer may point to any memory region. For example, the following function gets a signed 8-bit integer addressed by the void pointer “ptr” and the integer “offset”:

```
int ua_get_s8(void *ptr, unsigned int offset);
```

This function resolves the void pointer to the appropriate memory region (SRAM, DRAM, LOCAL_MEM, or Scratch). In almost all situations, you should use these general functions to access unaligned data. The function names are shorter and you do not have to specify the memory region in which the data resides.

However, in rare cases, the compiler will be unable to resolve the pointer to a memory region. In these cases, you should use one of the memory-region-qualified unaligned functions. For example:

```
int ua_get_s8_dram(DRAM_VOID *p, unsigned int offset);
```

This function resolves the pointer to the DRAM memory region and returns a signed 8-bit integer from DRAM.

The following sections summarize all unaligned get, set, and memcpy intrinsics.

4.2.1 Unaligned Get Functions

Table 8 summarizes the unaligned get functions.

Table 8. Unaligned Get Functions (Sheet 1 of 3)

Name (args)	Description
int ua_get_s8(void *ptr, unsigned int offset);	Gets a signed 8-bit integer addressed by the pointer and offset from the appropriate memory region.
int ua_get_s16(void *ptr, unsigned int offset);	Gets a signed 16-bit integer addressed by the pointer and offset from the appropriate memory region.
int ua_get_s32(void *ptr, unsigned int offset);	Gets a signed 32-bit integer addressed by the pointer and offset from the appropriate memory region.

Table 8. Unaligned Get Functions (Continued) (Sheet 2 of 3)

Name (args)	Description
long long ua_get_s64(void *ptr, unsigned int offset);	Gets a signed 64-bit integer addressed by the pointer and offset from the appropriate memory region.
unsigned int ua_get_u8(void *ptr, unsigned int offset);	Gets an unsigned 8-bit integer addressed by the pointer and offset from the appropriate memory region.
unsigned int ua_get_u16(void *ptr, unsigned int offset);	Gets an unsigned 16-bit integer addressed by the pointer and offset from the appropriate memory region.
unsigned int ua_get_u32(void *ptr, unsigned int offset);	Gets an unsigned 32-bit integer addressed by the pointer and offset from the appropriate memory region.
unsigned long long ua_get_u64(void *ptr, unsigned int offset);	Gets an unsigned 64-bit integer addressed by the pointer and offset from the appropriate memory region.
int ua_get_s8_dram(DRAM_VOID *p, unsigned int offset);	Gets a signed 8-bit integer addressed by the pointer and offset from DRAM.
int ua_get_s8_sram(SRAM_VOID *p, unsigned int offset);	Gets a signed 8-bit integer addressed by the pointer and offset from SRAM.
int ua_get_s8_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets a signed 8-bit integer addressed by the pointer and offset from Scratch.
int ua_get_s8_lmem(LMEM_VOID *p, unsigned int offset);	Gets a signed 8-bit integer addressed by the pointer and offset from LOCAL_MEM.
int ua_get_s16_dram(DRAM_VOID *p, unsigned int offset);	Gets a signed 16-bit integer addressed by the pointer and offset from DRAM.
int ua_get_s16_sram(SRAM_VOID *p, unsigned int offset);	Gets a signed 16-bit integer addressed by the pointer and offset from SRAM.
int ua_get_s16_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets a signed 16-bit integer addressed by the pointer and offset from Scratch.
int ua_get_s16_lmem(LMEM_VOID *p, unsigned int offset);	Gets a signed 16-bit integer addressed by the pointer and offset from LOCAL_MEM.
int ua_get_s32_dram(DRAM_VOID *p, unsigned int offset);	Gets a signed 32-bit integer addressed by the pointer and offset from DRAM.
int ua_get_s32_sram(SRAM_VOID *p, unsigned int offset);	Gets a signed 32-bit integer addressed by the pointer and offset from SRAM.
int ua_get_s32_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets a signed 32-bit integer addressed by the pointer and offset from Scratch.
int ua_get_s32_lmem(LMEM_VOID *p, unsigned int offset);	Gets a signed 32-bit integer addressed by the pointer and offset from LOCAL_MEM.
long long ua_get_s64_dram(DRAM_VOID *p, unsigned int offset);	Gets a signed 64-bit integer addressed by the pointer and offset from DRAM.
long long ua_get_s64_sram(SRAM_VOID *p, unsigned int offset);	Gets a signed 64-bit integer addressed by the pointer and offset from SRAM.
long long ua_get_s64_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets a signed 64-bit integer addressed by the pointer and offset from Scratch.
long long ua_get_s64_lmem(LMEM_VOID *p, unsigned int offset);	Gets a signed 64-bit integer addressed by the pointer and offset from LOCAL_MEM.
unsigned int ua_get_u8_dram(DRAM_VOID *p, unsigned int offset);	Gets an unsigned 8-bit integer addressed by the pointer and offset from DRAM.
unsigned int ua_get_u8_sram(SRAM_VOID *p, unsigned int offset);	Gets an unsigned 8-bit integer addressed by the pointer and offset from SRAM.

Table 8. Unaligned Get Functions (Continued) (Sheet 3 of 3)

Name (args)	Description
unsigned int ua_get_u8_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets an unsigned 8-bit integer addressed by the pointer and offset from Scratch.
unsigned int ua_get_u8_lmem(LMEM_VOID *p, unsigned int offset);	Gets an unsigned 8-bit integer addressed by the pointer and offset from LOCAL_MEM.
unsigned int ua_get_u16_dram(DRAM_VOID *p, unsigned int offset);	Gets an unsigned 16-bit integer addressed by the pointer and offset from DRAM.
unsigned int ua_get_u16_sram(SRAM_VOID *p, unsigned int offset);	Gets an unsigned 16-bit integer addressed by the pointer and offset from SRAM.
unsigned int ua_get_u16_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets an unsigned 16-bit integer addressed by the pointer and offset from Scratch.
unsigned int ua_get_u16_lmem(LMEM_VOID *p, unsigned int offset);	Gets an unsigned 16-bit integer addressed by the pointer and offset from LOCAL_MEM.
unsigned int ua_get_u32_dram(DRAM_VOID *p, unsigned int offset);	Gets an unsigned 32-bit integer addressed by the pointer and offset from DRAM.
unsigned int ua_get_u32_sram(SRAM_VOID *p, unsigned int offset);	Gets an unsigned 32-bit integer addressed by the pointer and offset from SRAM.
unsigned int ua_get_u32_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets an unsigned 32-bit integer addressed by the pointer and offset from Scratch.
unsigned int ua_get_u32_lmem(LMEM_VOID *p, unsigned int offset);	Gets an unsigned 32-bit integer addressed by the pointer and offset from LOCAL_MEM.
unsigned long long ua_get_u64_dram(DRAM_VOID *p, unsigned int offset);	Gets an unsigned 64-bit integer addressed by the pointer and offset from DRAM.
unsigned long long ua_get_u64_sram(SRAM_VOID *p, unsigned int offset);	Gets an unsigned 64-bit integer addressed by the pointer and offset from SRAM.
unsigned long long ua_get_u64_scratch(SCRATCH_VOID *p, unsigned int offset);	Gets an unsigned 64-bit integer addressed by the pointer and offset from Scratch.
unsigned long long ua_get_u64_lmem(LMEM_VOID *p, unsigned int offset);	Gets an unsigned 64-bit integer addressed by the pointer and offset from LOCAL_MEM.

4.2.2 Unaligned Set Functions

Table 9 summarizes the unaligned set functions.

Table 9. Unaligned Set Functions Summary (Sheet 1 of 2)

Name (args)	Description
void ua_set_8(void *ptr, unsigned int offset, unsigned int value);	Sets an 8-bit integer to the address specified by ptr and offset in the appropriate memory region.
void ua_set_16(void *ptr, unsigned int offset, unsigned int value);	Sets a 16-bit integer to the address specified by ptr and offset in the appropriate memory region.
void ua_set_32(void *ptr, unsigned int offset, unsigned int value);	Sets a 32-bit integer to the address specified by ptr and offset in the appropriate memory region.
void ua_set_64(void *ptr, unsigned int offset, unsigned long long value);	Sets a 64-bit integer to the address specified by ptr and offset in the appropriate memory region.
void ua_set_8_dram(DRAM_VOID *p, unsigned int offset, unsigned int val);	Sets an 8-bit integer to the address specified by ptr and offset in DRAM.
void ua_set_8_sram(SRAM_VOID *p, unsigned int offset, unsigned int val);	Sets an 8-bit integer to the address specified by ptr and offset in SRAM.
void ua_set_8_scratch(SCRATCH_VOID *p, unsigned int offset, unsigned int val);	Sets an 8-bit integer to the address specified by ptr and offset in Scratch.
void ua_set_8_lmem(LMEM_VOID *p, unsigned int offset, unsigned int val);	Sets an 8-bit integer to the address specified by ptr and offset in LOCAL_MEM.
void ua_set_16_dram(DRAM_VOID *p, unsigned int offset, unsigned int val);	Sets an 16-bit integer to the address specified by ptr and offset in DRAM.
void ua_set_16_sram(SRAM_VOID *p, unsigned int offset, unsigned int val);	Sets an 16-bit integer to the address specified by ptr and offset in SRAM.
void ua_set_16_scratch(SCRATCH_VOID *p, unsigned int offset, unsigned int val);	Sets an 16-bit integer to the address specified by ptr and offset in Scratch.
void ua_set_16_lmem(LMEM_VOID *p, unsigned int offset, unsigned int val);	Sets an 16-bit integer to the address specified by ptr and offset in LOCAL_MEM.
void ua_set_32_dram(DRAM_VOID *p, unsigned int offset, unsigned int val);	Sets an 32-bit integer to the address specified by ptr and offset in DRAM.
void ua_set_32_sram(SRAM_VOID *p, unsigned int offset, unsigned int val);	Sets an 32-bit integer to the address specified by ptr and offset in SRAM.
void ua_set_32_scratch(SCRATCH_VOID *p, unsigned int offset, unsigned int val);	Sets an 32-bit integer to the address specified by ptr and offset in Scratch.
void ua_set_32_lmem(LMEM_VOID *p, unsigned int offset, unsigned int val);	Sets an 32-bit integer to the address specified by ptr and offset in LOCAL_MEM.
void ua_set_64_dram(DRAM_VOID *p, unsigned int offset, unsigned long long val);	Sets an 64-bit integer to the address specified by ptr and offset in DRAM.

Table 9. Unaligned Set Functions Summary (Continued) (Sheet 2 of 2)

Name (args)	Description
void ua_set_64_sram(SRAM_VOID *p, unsigned int offset, unsigned long long val);	Sets an 64-bit integer to the address specified by ptr and offset in SRAM.
void ua_set_64_scratch(SCRATCH_VOID *p, unsigned int offset, unsigned long long val);	Sets an 64-bit integer to the address specified by ptr and offset in Scratch.
void ua_set_64_lmem(LMEM_VOID *p, unsigned int offset, unsigned long long val);	Sets an 64-bit integer to the address specified by ptr and offset in LOCAL_MEM.

4.2.3 Unaligned Memory Copy Functions

Table 10 shows the unaligned memory copy functions

Table 10. Unaligned memcpy Functions (Sheet 1 of 2)

Name (args)	Description
<pre>void ua_memcpy(void *dst, unsigned int dst_off, void *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src to the dst. Both src and dst can reside in any memory region.</p>
<pre>void ua_memcpy_dram_dram(DRAM_VOID *dst, unsigned int dst_off, DRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src to the dst. Both src and dst must reside in DRAM.</p>
<pre>void ua_memcpy_dram_sram(DRAM_VOID *dst, unsigned int dst_off, SRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in SRAM to the dst in DRAM.</p>
<pre>void ua_memcpy_dram_scratch(DRAM_VOID *dst, unsigned int dst_off, SCRATCH_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in Scratch to the dst in DRAM.</p>
<pre>void ua_memcpy_dram_lmem(DRAM_VOID *dst, unsigned int dst_off, LMEM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in LOCAL_MEM to the dst in DRAM.</p>
<pre>void ua_memcpy_sram_dram(SRAM_VOID *dst, unsigned int dst_off, DRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in DRAM to the dst in SRAM.</p>
<pre>void ua_memcpy_sram_sram(SRAM_VOID *dst, unsigned int dst_off, SRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src to the dst. Both src and dst must reside in SRAM.</p>
<pre>void ua_memcpy_sram_scratch(SRAM_VOID *dst, unsigned int dst_off, SCRATCH_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in Scratch to the dst in SRAM.</p>
<pre>void ua_memcpy_sram_lmem(SRAM_VOID *dst, unsigned int dst_off, LMEM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in LOCAL_MEM to the dst in SRAM.</p>

Table 10. Unaligned memcpy Functions (Continued) (Sheet 2 of 2)

Name (args)	Description
<pre>void ua_memcpy_scratch_dram(SCRATCH_VOID *dst, unsigned int dst_off, DRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in DRAM to the dst in Scratch.</p>
<pre>void ua_memcpy_scratch_sram(SCRATCH_VOID *dst, unsigned int dst_off, SRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in SRAM to the dst in Scratch.</p>
<pre>void ua_memcpy_scratch_scratch(SCRATCH_VOID *dst, unsigned int dst_off, SCRATCH_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src to the dst. Both src and dst must reside in Scratch.</p>
<pre>void ua_memcpy_scratch_lmem(SCRATCH_VOID *dst, unsigned int dst_off, LMEM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in LOCAL_MEM to the dst in Scratch.</p>
<pre>void ua_memcpy_lmem_dram(LMEM_VOID *dst, unsigned int dst_off, DRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in DRAM to the dst in LOCAL_MEM.</p>
<pre>void ua_memcpy_lmem_sram(LMEM_VOID *dst, unsigned int dst_off, SRAM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in SRAM to the dst in LOCAL_MEM.</p>
<pre>void ua_memcpy_lmem_scratch(LMEM_VOID *dst, unsigned int dst_off, SCRATCH_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src in Scratch to the dst in LOCAL_MEM.</p>
<pre>void ua_memcpy_lmem_lmem(LMEM_VOID *dst, unsigned int dst_off, LMEM_VOID *src, unsigned int src_off, unsigned int length);</pre>	<p>Copies the number of bytes specified in the length argument from the src to the dst. Both src and dst must reside in LOCAL_MEM.</p>

4.3 Memory I/O Functions

The memory and I/O functions provide direct access to the machine instructions Scratch, SRAM, DRAM, PCI, MSF, and REFLECT. The data for memory and I/O operations is generally one or more longwords or quadwords. The arguments for the data are specified as `void *` to allow you to pass any structure or array of the appropriate size for these operations. There are restrictions placed upon the data argument as described at the end of this chapter (Refer to [Section 4.10, “Restrictions On Ininsics” on page 354](#)). These restrictions are needed to satisfy the transfer register operand restrictions imposed by the microcode underlying the intrinsic and to deal with the asynchronous programming model exposed by the microcode. Take special care to properly call the `__free_write_buffer()` intrinsic when performing an asynchronous memory write operation (i.e. a write that waits on “sig_done”). Without a call to this intrinsic, the compiler may prematurely reuse the write transfer registers involved in the operation before the write has completed.

For those intrinsics that accept `SIGNAL_PAIR`, the `sync` argument must be `sig_done`; therefore, the caller must wait for the signal using `__wait_for_all()`, `__wait_for_any()`, or `signal_test()`, etc. Also, asynchronous memory reads may require the use of `__implicit_read()` if not all the data is used. See [Section 7.2, “Things to Remember When Writing Microengine C Code” on page 374](#) for details.

4.3.1 Transfer Register Modifiers

The following `__declspec` modifiers are used in most memory and CSR accessing intrinsics to help you cope with the restrictions detailed in [Section 4.10](#). You receive an error if you don't specify a `declspec` modifier to an intrinsic that expects one. If you pass a parameter not annotated with the appropriate `declspec` modifier, the compiler gives both a warning and an error. For example, if an intrinsic is expecting an `sram_read_reg` and you pass it an `sram_write_reg`, you receive a warning and a compiler error.

```
__declspec (sram_read_reg)
__declspec (sram_write_reg)
__declspec (dram_read_reg)
__declspec (dram_write_reg)
```

Some intrinsics expect constant parameters (i.e., `count`, `sync`). For `count`, the compiler now tries to generate an indirect reference if a non-constant is passed and will run with a loss of performance. For `sync` or `csr`, etc., however, the compiler reports an error if an intrinsic that expects a constant doesn't receive one.

4.3.2 Memory I/O Data types

This section describes the data types used with the Memory I/O functions. [Table 11](#) summarizes these data types.

Table 11. Memory I/O Data Types (Sheet 1 of 2)

Data Type	Description
<code>bytes_specifier_t</code>	This enumeration type specifies the bytes to be used with crc operations.
<code>cap_csr_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on CAP CSR read/write operations on SRAM channel CSRs with the <code>indirect_ref</code> attribute.
<code>cap_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on CAP with the <code>indirect_ref</code> attribute.
<code>dram_rbuf_tbuf_ind_t</code>	Structure that provides additional or overriding qualifiers on read/write operations between the receive/transmit FIFO and DRAM memory.
<code>dram_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on read/write operations on DRAM memory with the <code>indirect_ref</code> attribute.
<code>generic_ind_t</code>	Union of all indirect qualifier data types and an unsigned int value field that allows you to set or clear an entire indirect qualifier with one assignment rather than setting each field individually.
<code>hash_ind_t</code>	Structure that provide additional or overriding qualifiers on hash operations with the <code>indirect_ref</code> attribute.
<code>msf_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on MSF (Media Switch Fabric) operations with the <code>indirect_ref</code> attribute.
<code>pci_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on PCI read/write operations with the <code>indirect_ref</code> attribute.
<code>reflect_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on Reflector operations with the <code>indirect_ref</code> attribute.
<code>reflect_sig_t</code>	This enumeration type specifies the type of signal for reflect operation
<code>scratch_atomic_ind_t</code>	Structure that provides additional or overriding qualifiers on atomic operations on Scratch memory with the <code>indirect_ref</code> attribute.
<code>scratch_read_write_ind_t</code>	Structure that provides additional or overriding qualifiers on read/write operations on Scratch memory with the <code>indirect_ref</code> attribute.
<code>scratch_ring_ind_t</code>	Structure that provides additional or overriding qualifiers on get/put operations on Scratch memory with the <code>indirect_ref</code> attribute.
<code>sram_atomic_ind_t</code>	Structure that provides additional or overriding qualifiers on atomic operations on SRAM memory with the <code>indirect_ref</code> attribute.

Table 11. Memory I/O Data Types (Continued) (Sheet 2 of 2)

Data Type	Description
sram_csr_read_write_ind_t	Structure that provides additional or overriding qualifiers on CSR read/write operations on SRAM channel CSRs with the indirect_ref attribute.
sram_dequeue_ind_t	Structure that provides additional or overriding qualifiers on dequeue operations on SRAM memory.
sram_enqueue_ind_t	Structure that provides additional or overriding qualifiers on enqueue operations on SRAM memory.
sram_journal_ind_t	Structure that provides additional or overriding qualifiers on Journal operations on SRAM memory with the indirect_ref attribute.
sram_read_qdesc_ind_t	Structure that provides additional or overriding qualifiers on Read Queue Descriptor operations on SRAM memory with the indirect_ref attribute.
sram_read_write_ind_t	Structure that provides additional or overriding qualifiers on read/write operations on SRAM memory operations with the indirect_ref attribute.
sram_ring_ind_t	Structure that provides additional or overriding qualifiers on GET/PUT on SRAM memory with the indirect_ref attribute.
sync_t	Enumeration used to specify the synchronization option for memory I/O operations.

4.3.2.1 sync_t

This enumeration type specifies the synchronization option to be used with memory or I/O operations.

Field Name	Description
ctx_swap	Swap out until operation is complete.
sig_done	Continue operation; set signal when operation is complete.



4.3.2.2 bytes_specifier_t

This enumeration type specifies the bytes to be used with CRC operations.

Field Name	Description	
	Big Endian	Little Endian
bytes_0_3	0, 1, 2, 3	3, 2, 1, 0
bytes_0_2	0, 1, 2	2, 1, 0
bytes_0_1	0, 1	1, 0
byte_0	0	0
bytes_1_3	1, 2, 3	3, 2, 1
bytes_2_3	2, 3	3, 2
byte_3	3	3

4.3.2.3 `reflect_signal_t`

This enumeration type specifies the type of signal for reflect operation.

Field Name	Description
<code>sig_initiator</code>	Signal current microengine.
<code>sig_remote</code>	Signal remote microengine.
<code>sig_both</code>	Signal both microengines.

4.3.2.4 pci_read_write_ind_t, sram_read_write_ind_t

These two structures contain identical fields and provide additional or overriding qualifiers on read/write operations in PCI and SRAM memory respectively with the indirect_ref attribute. For further details on this indirect qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. These structures contain the following unsigned int bitfields.

Field Name	Size	Description
ctx	3	Specifies context where result will be written and signaled upon completion.
ov_ctx	1	1 to use the context specified in the ctx field; 0 to use the current context.
ov_xadd	1	1 to override the transfer register (this should never be set).
xadd	7	The starting transfer register.
byte_mask	4	Mask of bytes to read or write.
reserved	4	Unused.
ov_byte_mask	1	1 to use the byte mask specified in the byte_mask field; 0 to use the default (all bytes read or written).
ref_count	4	Reference count indicating the number of longwords to read or write. The value encoded in this field is one less than the reference count. Hence, valid values are 0 - 15.
ov_ref_count	1	1 to use the count specified in the ref_count field; 0 to use the count argument to the function.
ueng_addr	5	Specifies the microengine where the result is to be written and signaled upon completion.
ov_ueng_addr	1	1 to use the microengine specified in the ueng_addr field; 0 to use the issuing microengine (this should never be set)

Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the ov_xadd field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the ov_ueng_addr fields.

4.3.2.5 `scratch_read_write_ind_t`, `scratch_ring_ind_t`, `sram_read_qdesc_ind_t`, `sram_ring_ind_t`, `sram_journal_ind_t`, `cap_read_write_ind_t`, `msf_read_write_ind_t`, `reflect_read_write_ind_t`

These eight structures contain identical fields and provide additional or overriding qualifiers on read/write operations on their associated components of the IXP2XXX Network Processor using the `indirect_ref` attribute.

The `scratch_read_write_ind_t` structure provides qualifiers for read and write operations in Scratch memory.

The `scratch_ring_ind_t` structure provides qualifiers for get/put operations in Scratch memory.

The `sram_read_qdesc_ind_t` structure provides qualifiers on Read Queue Descriptor operations.

The `sram_ring_ind_t` structure provides qualifiers on GET/PUT operations.

The `sram_journal_ind_t` structure provides qualifiers for Journal operations.

The `msf_read_write_ind_t` structure provides qualifiers for read and write operations on MSF (Media Switch Fabric) operations.

The `reflect_read_write_ind_t` structure provides qualifiers for read and write operations on Reflector operations.

The `cap_read_write_ind_t` structure provides additional or overriding qualifiers on CAP with the `indirect_ref` attribute.

For further details on the `indirect_ref` qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. These structures contain the following unsigned int bitfields.

Field Name	Size	Description
<code>ov_ueng_addr</code>	1	1 to use the microengine specified in the <code>ueng_addr</code> field; 0 to use the issuing microengine (this should never be set)
<code>ueng_addr</code>	5	Specifies the microengine where the result is to be written and signaled upon completion.
<code>ov_ref_count</code>	1	1 to use the count specified in the <code>ref_count</code> field; 0 to use the count argument to the function.
<code>ref_count</code>	4	Reference count indicating the number of longwords to read or write. The value encoded in this field is one less than the reference count. Hence, valid values are 0 - 15.
<code>reserved</code>	9	Unused.
<code>xadd</code>	7	The starting transfer register.
<code>ov_xadd</code>	1	1 to override the transfer register (this should never be set).
<code>ov_ctx</code>	1	1 to use the context specified in the <code>ctx</code> field; 0 to use the current context.
<code>ctx</code>	3	Specifies context where result will be written and signaled upon completion.



Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the `ov_xadd` field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the `ov_ueng_addr` fields.

4.3.2.6 dram_read_write_ind_t

This structure provides additional or overriding qualifiers on read/write operations on DRAM memory with the indirect_ref attribute. For further details on this indirect qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This structure contains the following unsigned int bitfields.

Field Name	Size	Description
ctx	3	Specifies context where result will be written and signaled upon completion.
ov_ctx	1	1 to use the context specified in the ctx field; 0 to use the current context.
ov_xadd	1	1 to override the transfer register (this should never be set).
xadd	7	The starting transfer register.
byte_mask	8	Mask of bytes to read or write.
ov_byte_mask	1	1 to use the byte mask specified in the byte_mask field; 0 to use the default (all bytes read or written).
ref_count	4	Reference count indicating the number of longwords to read or write. The value encoded in this field is one less than the reference count. Hence, valid values are 0 - 15.
ov_ref_count	1	1 to use the count specified in the ref_count field; 0 to use the count argument to the function.
ueng_addr	5	Specifies ME where result will be written and signaled upon completion.
ov_ueng_addr	1	1 to use the microengine specified in the ueng_addr field; 0 to use the issuing microengine (this should never be set)

Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the ov_xadd field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the ov_ueng_addr fields.

See [Section 3.1.10.1, “Compiler Limitations of Endian Support.”](#) on page 38 for pitfalls when using ov_byte_mask in big-endian mode.

4.3.2.7 sram_atomic_ind_t

This structure provides additional or overriding qualifiers on atomic operations on SRAM memory with the `indirect_ref` attribute. For further details on this indirect qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This structure contains the following unsigned int bitfields.

Field Name	Size	Description
<code>ctx</code>	3	Specifies context where result will be written and signaled upon completion.
<code>ov_ctx</code>	1	1 to use the context specified in the <code>ctx</code> field; 0 to use the current context.
<code>ov_xadd</code>	1	1 to override the transfer register (this should never be set).
<code>xadd</code>	7	The starting transfer register.
<code>byte_mask</code>	4	Mask of bytes to read or write.
<code>reserved1</code>	4	Unused.
<code>ov_byte_mask</code>	1	1 to use the byte mask specified in the <code>byte_mask</code> field; 0 to use the default (all bytes read or written).
<code>reserved2</code>	5	Unused.
<code>ueng_addr</code>	5	Specifies ME where result will be written and signaled upon completion.
<code>ov_ueng_addr</code>	1	1 to use the microengine specified in the <code>ueng_addr</code> field; 0 to use the issuing microengine (this should never be set).

Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the `ov_xadd` field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the `ov_ueng_addr` fields.

4.3.2.8 `scratch_atomic_ind_t`, `sram_csr_read_write_ind_t`, `cap_csr_read_write_ind_t`

These three structures provide additional or overriding qualifiers on atomic operations on Scratch memory and CSR read/write operations on SRAM channel CSRs respectively with the `indirect_ref` attribute. For further details on this indirect qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This structure contains the following unsigned int bitfields.

Field Name	Size	Description
<code>ctx</code>	3	Specifies context where result will be written and signaled upon completion.
<code>ov_ctx</code>	1	1 to use the context specified in the <code>ctx</code> field; 0 to use the current context.
<code>ov_xadd</code>	1	1 to override the transfer register (this should never be set).
<code>xadd</code>	7	The starting transfer register.
<code>reserved</code>	14	Unused.
<code>ueng_addr</code>	5	Specifies ME where result will be written and signaled upon completion.
<code>ov_ueng_addr</code>	1	1 to use the microengine specified in the <code>ueng_addr</code> field; 0 to use the issuing microengine (this should never be set).

Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the `ov_xadd` field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the `ov_ueng_addr` fields.

4.3.2.9 dram_rbuf_tbuf_ind_t

This structure provides additional or overriding qualifiers on read and write operations between the receive and transmit FIFO and DRAM memory. For further details on this indirect qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This structure contains the following unsigned int bitfields.

Field Name	Size	Description
ctx	3	Specifies context where result will be written and signaled upon completion.
ov_ctx	1	1 to use the context specified in the ctx field; 0 to use the current context.
ov_buf_addr	1	1 to override the rbuf/tbuf address.
buf_addr	14	rbuf/tbuf address.
reserved	2	Unused.
ref_count	4	Number of quadwords to read or write.
ov_ref_count	1	1 to use the count specified in the ref_count field; 0 to use the count argument to the functions.
ueng_addr	5	Specifies ME where result will be written and signaled upon completion.
ov_ueng_addr	1	1 to use the microengine specified in the ueng_addr field; 0 to use the issuing microengine (this should never be set).

Note: Since the compiler does all register allocation (including all transfer registers) for each Microengine, it is an error to set the ov_ueng_addr fields to override the transfer register allocated (which the compiler doesn't have knowledge of at compilation) and specified in the instruction by the compiler.

4.3.2.10 sram_enqueue_ind_t

This struct provides additional or overriding qualifiers on enqueue operations on SRAM memory.

Field Name	Size	Description
reserved1	12	Unused.
seg_count	6	Segment count.
sop	1	Set SOP bit in Q_link.
eop	1	Set EOP bit in Q_link.
ov_eop_sop_seg_count	1	1 to use the seg_count, sop, and eop above
reserved2	11	Unused.

4.3.2.11 sram_dequeue_ind_t

This struct provides additional or overriding qualifiers on dequeue operations on SRAM memory.

Field Name	Size	Description
ctx	3	Specifies context where result will be written and signaled upon completion.
ov_ctx	1	1 to use the context above, 0 to use the current context.
reserved	22	Unused.
ueng_addr	5	Specifies ME where result will be written and signaled upon completion.
ov_ueng_addr	1	1 to use ueng_addr above, 0 to use the current ueng (this should never be set).

Note: Since the compiler does all register allocation (including all transfer registers) for each Microengine, it is an error to set the ov_ueng_addr fields to override the transfer register allocated (which the compiler doesn't have knowledge of at compilation) and specified in the instruction by the compiler.

4.3.2.12 hash_ind_t

Structure that provide additional or overriding qualifiers on hash operations with the indirect_ref attribute.

Field Name	Size	Description
ctx	3	Specifies context where result will be written and signaled upon completion.
ov_ctx	1	1 to use the context above, 0 to use the current context.
ov_xadd	1	1 to override the transfer register (this should never be set)
xadd	7	The starting transfer register.
reserved1	9	Unused.
hash_count	2	Hash count 1, 2, or 3.
reserved2	2	Unused.
ov_ref_count	1	1 to use the count above, 0 to use the count argument to the function.
ueng_addr	5	Specifies ME where result will be written and signaled upon completion.
ov_ueng_addr	1	1 to use ueng_addr above, 0 to use the current ueng_addr (this should never be set).

Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the ov_xadd field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the ov_ueng_addr fields.

4.3.2.13 generic_ind_t

This type is a union of an unsigned integer with all the indirect qualifier data types. By setting/clearing the unsigned integer field "value" you can set/clear the entire indirect qualifier with one assignment as opposed to setting each field individually.

```
typedef union
{
    pci_read_write_ind_t          pci_rw;
    sram_read_write_ind_t         sram_rw;
    scratch_read_write_ind_t      scratch_rw;
    dram_read_write_ind_t         dram_rw;
    sram_atomic_ind_t             sram_atomic;
    scratch_atomic_ind_t          scratch_atomic;
    dram_rbuf_tbuf_ind_t          dram_rbuf_tbuf;
    sram_csr_read_write_ind_t     sram_csr_rw;
    scratch_ring_ind_t            scratch_ring;
    sram_read_qdesc_ind_t         sram_rd_qdesq;
    sram_journal_ind_t            sram_journal;
    cap_read_write_ind_t          csr_rw;
    msf_read_write_ind_t          msf_rw;
    reflect_read_write_ind_t      reflect_rw;
    sram_ring_ind_t               sram_ring;
    sram_enqueue_ind_t            sram_enqueue;
    sram_dequeue_ind_t            sram_dequeue;
    hash_ind_t                     hash;
    reflect_read_write_ind_t      reflect_rw;
    unsigned int                   value;
} generic_ind_t;
```

4.3.3 Memory I/O Functions

This section describes the Memory I/O functions. These functions are divided into the following categories.

- Scratch Operations
- SRAM Operations
- DRAM Operations
- Media Switch Fabric (MSF) Operations
- PCI Operations
- Reflection Operations
- Generic Operations

4.3.3.1 Scratch Operations

The functions described in this section perform I/O operations in Scratch memory. [Table 12](#) summarizes these functions.

Table 12. Scratch Operation Summary (Sheet 1 of 4)

Name (args)	Description
void scratch_read[_D](__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(scratch) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);	Reads count longwords from Scratch RAM at the specified address into the transfer register specified by data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is available only in 8-context mode.
void scratch_read[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(scratch) *address, unsigned int max_nn, scratch_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);	Reads up to max_nn longwords from Scratch RAM at the specified address into the transfer register specified by data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is available only in 8-context mode. The ind argument provides additional parameters and overrides.
void scratch_write[_D](__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(scratch) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);	Writes count longwords to Scratch RAM at the specified address from the transfer register specified by data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode.
void scratch_write[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(scratch) *address, unsigned int max_nn, scratch_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);	Writes up to max_nn longwords to Scratch RAM at the specified address from the transfer register specified by data. The ind argument provides additional parameters and overrides. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode.
void scratch_incr(volatile void __declspec(scratch) *address);	Increments the longword in Scratch RAM at the specified address by one.
void scratch_incr_ind(volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind);	Increments the longword in Scratch RAM at the specified address by one. The ind argument provides additional parameters and overrides.
void scratch_decr(volatile void __declspec(scratch) *address);	Decrements the longword in Scratch RAM at the specified address by one.
void scratch_decr_ind(volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind);	Decrements the longword in Scratch RAM at the specified address by one. The ind argument provides additional parameters and overrides.
void scratch_add[_D](__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL *sig_ptr);	Increments the longword in Scratch RAM at the specified address by the value specified in data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode.
void scratch_add[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);	Increments the longword in Scratch RAM at the specified address by the value specified in data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode. The ind argument provides additional parameters and overrides.

Table 12. Scratch Operation Summary (Continued) (Sheet 2 of 4)

Name (args)	Description
<pre>void scratch_sub[_D](__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Decrements the longword in Scratch RAM at the specified address by the value specified in data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_sub[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Decrements the longword in Scratch RAM at the specified address by the value specified in data. The <code>ind</code> argument provides additional parameters and overrides. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_set_bits[_D](__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Sets the bits in the specified mask in the longword at address in Scratch ram. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_set_bits[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, scratch_atomic_ind ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Sets the bits in the specified mask in the longword at address in Scratch ram. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void scratch_clear_bits[_D](__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Clears the bits in the specified mask in the longword at address in Scratch ram. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_clear_bits[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, scratch_atomic_ind ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Clears the bits in the specified mask in the longword at address in Scratch ram. The <code>ind</code> argument provides additional parameters and overrides. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_test_and_set_bits[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into <code>val</code> the initial value of the longword at address in Scratch ram. It then sets the bits specified in mask in the longword at address in Scratch ram. The <code>_D</code> version of this intrinsic must be used if the <code>val</code> and mask arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_test_and_set_bits[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into <code>val</code> the initial value of the longword at address in Scratch ram. It then sets the bits specified in mask in the longword at address in Scratch ram. The <code>ind</code> argument provides additional parameters and overrides. The <code>_D</code> version of this intrinsic must be used if the <code>val</code> and mask arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>

Table 12. Scratch Operation Summary (Continued) (Sheet 3 of 4)

Name (args)	Description
<pre>void scratch_test_and_clear_bits[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in Scratch ram. It then clears the bits specified in mask in the longword at address in Scratch ram. The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_test_and_clear_bits_ind[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in Scratch ram. It then clears the bits specified in mask in the longword at address in Scratch ram. The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_test_and_add[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in Scratch ram. It then increments the longword at address by the value specified by data. The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_test_and_add[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in Scratch ram. It then increments the longword at address by the value specified by data. The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_test_and_sub[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in Scratch ram. It then decrements the longword at address by the value specified by data. The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_test_and_sub[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in Scratch ram. It then decrements the longword at address by the value specified by data. The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfers registers. The _D version is only available on IXP28XX Rev B. hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_test_and_incr[_D](__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads the initial value of the longword at address in Scratch RAM into the transfer register specified by val. It then increments the longword at address in Scratch RAM by one. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>

Table 12. Scratch Operation Summary (Continued) (Sheet 4 of 4)

Name (args)	Description
<pre>void scratch_test_and_incr[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads the initial value of the longword at address in Scratch RAM into the transfer register specified by val. It then increments the longword at address in Scratch RAM by one. The <code>_D</code> version of this intrinsic must be used if the val argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_test_and_decr[_D](__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads the initial value of the longword at address in Scratch RAM into the transfer register pointed to by val. It then decrements the longword at address in Scratch RAM by one. The <code>_D</code> version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>
<pre>void scratch_test_and_decr[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads into the transfer register pointed to by val the initial value of the longword at address in Scratch ram. It then decrements the longword at address in Scratch RAM by one. The <code>_D</code> version of this intrinsic must be used if the val argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_swap[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address. It then writes the value specified in data to the longword at address. The <code>_D</code> version of this intrinsic must be used if the val and data arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_swap[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, scratch_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address. It then writes the value specified in data to the longword at address. The <code>_D</code> version of this intrinsic must be used if the val and data arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_get_ring[_D](__declspec([sram, dram]_read_reg) unsigned int *data, volatile void __declspec(scratch) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Gets count longwords of data from the Scratch ring specified by address and returns it in the transfer register specified by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>
<pre>void scratch_get_ring[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *data, volatile void __declspec(scratch) *address, unsigned int max_nn, scratch_ring_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Gets up to max_nn longwords of data from the Scratch ring specified by address and returns it in the transfer register specified by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>
<pre>void scratch_put_ring[_D](__declspec(sram_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Puts count longwords from data into the Scratch ring specified by address. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode.</p>
<pre>void scratch_put_ring[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(scratch) *address, unsigned int max_nn, scratch_ring_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Puts up to max_nn longwords from data into the Scratch ring specified by address. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX Rev B. hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>

4.3.3.1.1 scratch_read(), scratch_read_D()

Function Syntax:

```
void scratch_read[_D] (  
    __declspec([sram,dram]_read_reg) void * data,  
    volatile void __declspec(scratch) * address,  
    unsigned int count,  
    sync_t sync,  
    SIGNAL* sig_ptr);
```

Description:

These functions read count longwords from Scratch RAM at the specified address and place the data into the structure addressed by data. The scratch_read() intrinsic reads the data into an SRAM transfer register while the scratch_read_D() variant reads the data into a DRAM transfer register. The count argument must be in the range of 1 through 16.

Argument count is preferred to be a constant literal; otherwise, the compiler generates an indirect_ref resulting in a loss of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Address of data buffer to read into.
address	Address to read from.
count	Number of longwords to read/write in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.2 `scratch_read_ind()`, `scratch_read_D_ind()`

Function Syntax:

```
void scratch_read[_D]_ind(
    __declspec( [sram, dram] read_reg) void *data,
    volatile void __declspec(scratch) *address,
    unsigned int max_nn,
    scratch_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read up to `max_nn` longwords from Scratch RAM at the specified address and place the data into the structure addressed by `data`. The `scratch_read_ind()` function reads the data into an SRAM transfer register while the `scratch_read_D_ind` variant reads the data into a DRAM transfer register. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. There are restrictions on the value specified in the override as noted in the description of the `scratch_read_write_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `data` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Address of data buffer to read into.
<code>address</code>	Address to read from
<code>max_nn</code>	Number of longwords to read/write in the range of 1 - 16.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.3 `scratch_write()`, `scratch_write_D()`

Function Syntax:

```
void scratch_write[_D] (  
    __declspec([sram, dram]_write_reg) void *data,  
    volatile void __declspec(scratch) *address,  
    unsigned int count,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

These functions write count longwords to Scratch RAM at the specified address from the transfer register addressed by data. The `scratch_write()` function transfers the data from an SRAM transfer register while the `scratch_write_D` variant transfers the data from a DRAM transfer register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise the compiler generates an `indirect_ref`, resulting in a loss of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Address of data buffer to read from.
<code>address</code>	Address to write to.
<code>count</code>	Number of longwords to read and write in the range of 1 - 16.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.4 `scratch_write_ind()`, `scratch_write_D_ind()`

Function Syntax:

```
void scratch_write[_D]_ind(
    __declspec([sram, dram] write_reg) void *data,
    volatile void __declspec(scratch) *address,
    unsigned int max_nn,
    scratch_read_write_ind_t ind
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes up to `max_nn` longwords to Scratch RAM at the specified address from the structure addressed by `data`. The `scratch_write()` function transfers the data from an SRAM transfer register while the `scratch_write_D` variant transfers the data from a DRAM transfer register. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `scratch_read_write_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the `data` argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Address of data buffer to read from.
<code>address</code>	Address to write to.
<code>max_nn</code>	Number of longwords to read and write in the range of 1 - 16.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.



4.3.3.1.5 scratch_incr()

Function Syntax:

```
void scratch_incr(  
    volatile void __declspec(scratch) *address)
```

Description:

This function increments the longword in Scratch RAM at the specified address by one.

Arguments:

address	Address of the longword to increment.
---------	---------------------------------------

4.3.3.1.6 scratch_incr_ind()**Function Syntax:**

```
void scratch_incr_ind(  
    volatile void __declspec(scratch) *address),  
    scratch_atomic_ind_t ind);
```

Description:

This function increments the longword in Scratch RAM at the specified address by one. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type.

Arguments:

address	Address of the longword to increment.
ind	Indirect word.



4.3.3.1.7 scratch_decr()

Function Syntax:

```
void scratch_decr(  
    volatile void __declspec(scratch) *address)
```

Description:

This function decrements the longword in Scratch RAM at the specified address by one.

Arguments:

address	Address of longword to decrement.
---------	-----------------------------------

4.3.3.1.8 scratch_decr_ind()**Function Syntax:**

```
void scratch_decr_ind(  
    volatile void __declspec(scratch) *address),  
    scratch_atomic_ind_t ind);
```

Description:

This function decrements the longword in Scratch RAM at the specified address by one. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type.

Arguments:

address	Address of longword to decrement.
ind	Indirect word.

4.3.3.1.9 `scratch_add()`, `scratch_add_D()`

Function Syntax:

```
void scratch_add[_D] (  
    __declspec([sram, dram]_write_reg) void *data,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

This function increments the longword in Scratch RAM at the specified address by the argument data. The `sig_ptr` argument should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Value to add.
<code>address</code>	Address to read from.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.10 `scratch_add_ind()`, `scratch_add_D_ind()`

Function Syntax:

```
void scratch_add[_D]_ind(
    __declspec([sram, dram] write_reg) void *data,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function increments the longword in Scratch RAM RAM at the specified address by the argument data. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Value to add.
address	Address to read from.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.11 `scratch_sub()`, `scratch_sub_D()`

Function Syntax:

```
void scratch_sub[_D] (  
    __declspec([sram, dram]_write_reg) void *data,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

This function decrements the longword in Scratch RAM at the specified address by the argument data. The `sig_ptr` argument should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Value to subtract.
<code>address</code>	Address to read from.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.12 `scratch_sub_ind()`, `scratch_sub_D_ind()`

Function Syntax:

```
void scratch_sub[_D]_ind(
    __declspec([sram, dram] write_reg) void *data,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function decrements the longword in Scratch RAM at the specified address by the argument data. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Value to subtract.
address	Address to read from.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.13 `scratch_set_bits()`, `scratch_set_bits_D()`

Function Syntax:

```
void scratch_set_bits[_D](  
    __declspec([sram, dram]_write_reg) unsigned int *mask,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

This function sets the bits in the specified mask in the longword at address in Scratch ram. The `scratch_set_bits()` function obtains the mask from an SRAM transfer register while the `scratch_set_bits_D()` variant obtains the mask from a DRAM transfer register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>mask</code>	Mask to set.
<code>address</code>	Address to write.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.14 `scratch_set_bits_ind()`, `scratch_set_bits_D_ind()`

Function Syntax:

```
void scratch_set_bits[_D]_ind(
    __declspec([sram, dram] write_reg) unsigned int *mask,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function sets the bits in the specified mask in the longword at address in Scratch ram. The `scratch_set_bits()` function obtains the mask from an SRAM transfer register while the `scratch_set_bits_D()` variant obtains the mask from a DRAM transfer register. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `scratch_atomic_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

mask	Mask to set.
address	Address to write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.15 `scratch_clear_bits()`, `scratch_clear_bits_D()`

Function Syntax:

```
void scratch_clear_bits[_D] (  
    __declspec([sram, dram]_write_reg) unsigned int *mask,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

This function clears the bits in the specified mask in the longword at address in Scratch ram. The `scratch_clear_bits()` function obtains the mask from an SRAM transfer register while the `scratch_clear_bits_D()` variant obtains the mask from a DRAM transfer register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>mask</code>	Mask to clear.
<code>address</code>	Address to write.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.16 `scratch_clear_bits_ind()`, `scratch_clear_bits_D_ind()`

Function Syntax:

```
void scratch_clear_bits[_D]_ind(
    __declspec([sram, dram] write_reg) unsigned int *mask,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function clears the bits in the specified mask in the longword at address in Scratch ram. The `scratch_clear_bits_ind()` function obtains the mask from an SRAM transfer register while the `scratch_clear_bits_D_ind()` variant obtains the mask from a DRAM transfer register. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `scratch_atomic_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

mask	Mask to clear.
address	Address to write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.17 `scratch_test_and_set_bits()`, `scratch_test_and_set_bits_D()`

Function Syntax:

```
void scratch_test_and_set_bits[_D](  
    __declspec([sram, dram]_read_reg) unsigned int *val  
    __declspec([sram, dram]_write_reg) unsigned int *mask,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then sets the bits specified in the buffer mask in the longword at address in Scratch ram. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.18 `scratch_test_and_set_bits_ind()`, `scratch_test_and_set_bits_D_ind()`

Function Syntax:

```
void scratch_test_and_set_bits[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then sets the bits specified in the buffer mask in the longword at address in Scratch ram. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used when both the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.19 `scratch_test_and_clear_bits()`, `scratch_test_and_clear_bits_D()`

Function Syntax:

```
void scratch_test_and_clear_bits[_D](  
    __declspec([sram, dram]_read_reg) unsigned int *val  
    __declspec([sram, dram]_write_reg) unsigned int *mask,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then clears the bits specified in the buffer mask in the longword at address in Scratch ram. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.20 `scratch_test_and_clear_bits_ind()`, `scratch_test_and_clear_bits_D_ind()`

Function Syntax:

```
void scratch_test_and_clear_bits[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then clears the bits specified in the buffer mask in the longword at address in Scratch ram. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used when both the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.21 `scratch_test_and_add()`, `scratch_test_and_add_D()`

Function Syntax:

```
void scratch_test_and_add[_D] (  
    __declspec([sram, dram]_read_reg) unsigned int *val  
    __declspec([sram, dram]_write_reg) unsigned int *data,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then increments the longword at address in Scratch RAM by the argument data. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to add.
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.22 `scratch_test_and_add_ind()`, `scratch_test_and_add_D_ind()`

Function Syntax:

```
void scratch_test_and_add[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by data. It then increments the longword at address in Scratch RAM by the argument data. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used when both the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to add.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.23 `scratch_test_and_sub()`, `scratch_test_and_sub_D()`

Function Syntax:

```
void scratch_test_and_sub[_D] (  
    __declspec([sram, dram]_read_reg) unsigned int *val  
    __declspec([sram, dram]_write_reg) unsigned int *data,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then decrements the longword at address in Scratch RAM by the argument data. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to decrement
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.24 `scratch_test_and_sub[_D]_ind()`, `scratch_test_and_sub_D_ind()`

Function Syntax:

```
void scratch_test_and_sub[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then decrements the longword at address in Scratch RAM by the argument data. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used when both the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to decrement.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.25 `scratch_test_and_incr()`, `scratch_test_and_incr_D`

Function Syntax:

```
void scratch_test_and_incr[_D](
    __declspec([sram, dram]_read_reg) unsigned int *val
    volatile void __declspec(scratch) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions load the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. They then increment the longword at address in Scratch RAM by one. The `scratch_test_and_incr()` function loads the value into an SRAM register while the `scratch_test_and_incr_D()` function loads the value into a DRAM register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `val` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>val</code>	Value before write.
<code>address</code>	Address to read/write.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.26 `scratch_test_and_incr_ind()`, `scratch_test_and_incr_D_ind()`

Function Syntax:

```
void scratch_test_and_incr[_D]_ind(
    __declspec([sram, dram] read_reg) unsigned int *val
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions load the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. They then increment the longword at address in Scratch RAM by one. The `scratch_test_and_incr_ind()` function loads the value into an SRAM register while the `scratch_test_and_incr_D_ind()` function loads the value into a DRAM register. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `scratch_atomic_ind_t` data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

val	Value before write.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.27 `scratch_test_and_decr()`, `scratch_test_and_decr_D()`

Function Syntax:

```
void scratch_test_and_decr[_D] (  
    __declspec([sram, ram] read_reg) unsigned int *val  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

These two functions load the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. They then decrement the longword at address in Scratch RAM by one. The `scratch_test_and_decr()` function loads the value into an SRAM register while the `scratch_test_and_decr_D()` function loads the value into a DRAM register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `val` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>val</code>	Value before write.
<code>address</code>	Address to read/write.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.28 `scratch_test_and_decr_ind()`, `scratch_test_and_decr_D_ind()`

Function Syntax:

```
void scratch_test_and_decr[_D]_ind(
    __declspec([sram, dram] read_reg) unsigned int *val
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These two functions load the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. They then decrement the longword at address in Scratch RAM by one. The `scratch_test_and_decr_ind()` function loads the value into an SRAM register while the `scratch_test_and_decr_D_ind()` function loads the value into a DRAM register. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `scratch_atomic_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `val` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>val</code>	Value before write.
<code>address</code>	Address to read/write.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.1.29 `scratch_swap()`, `scratch_swap_D()`

Function Syntax:

```
void scratch_swap[_D] (  
    __declspec([sram, dram]_read_reg) unsigned int *val,  
    __declspec([sram, dram]_write_reg) unsigned int *data,  
    volatile void __declspec(scratch) *address,  
    sync_t sync,  
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then writes the value in buffer data to the longword at address in Scratch ram. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to add.
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.30 `scratch_swap_ind()`, `scratch_swap_D_ind()`

Function Syntax:

```
void scratch_swap[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val,
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(scratch) *address,
    scratch_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in Scratch RAM into the data buffer pointed to by val. It then writes the value in buffer data to the longword at address in Scratch RAM. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to add.
address	Address to read/write.
ind	Indirect word
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.



4.3.3.1.31 `scratch_get_ring()`, `scratch_get_ring_D()`

Function Syntax:

```
void scratch_get_ring[_D](
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(scratch) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions get count longwords of data from the Scratch ring specified by address and write the data into the buffer pointed to by val. The `scratch_get_ring()` function writes the data into an SRAM register while the `scratch_get_ring_D()` function writes the data into a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref`, resulting in a loss of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Buffer to write contents of Scratch to.
address	Address of the Scratch ring.
count	Number of longwords to read from the Scratch ring and write into val in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.32 scratch_get_ring_ind(), scratch_get_ring_D_ind()

Function Syntax:

```
void scratch_get_ring[_D]_ind(
    __declspec([sram, dram] read_reg) void *data,
    volatile void __declspec(scratch) *address,
    unsigned int max_nn,
    scratch_ring_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions are the indirect reference form of the scratch_ring_get function. They get up to max_nn longwords of data from the Scratch ring specified by address and write the data into the buffer pointed to by val. The scratch_get_ring_ind() function writes the data into an SRAM register while the scratch_get_ring_D_ind() function writes the data into a DRAM register. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_ring_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Buffer to write contents of Scratch to.
address	Address of the Scratch ring.
max_nn	Number of longwords to read from Scratch ring and write into val in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.33 scratch_put_ring(), scratch_put_ring_D()

Function Syntax:

```
void scratch_put_ring[_D](  
    __declspec([sram, dram]_read_reg) unsigned int *data,  
    volatile void __declspec(scratch) *address,  
    unsigned int count,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

This function puts count longwords of data from the buffer pointed to by the data argument into the Scratch ring specified by address. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref, resulting in a loss of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Buffer to read from.
address	Address of the Scratch ring to write to.
count	Number of longwords to read from buffer val and write into the Scratch ring pointed to by address in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.1.34 scratch_put_ring_ind(), scratch_put_ring_D_ind()

Function Syntax:

```
void scratch_put_ring[_D]_ind(
    __declspec( [sram, dram] read_reg) unsigned int *data,
    volatile void __declspec(scratch) *address,
    unsigned int max_nn,
    scratch_ring_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function is the indirect reference form of the scratch_ring_put function. It puts up to max_nn longwords of data from the buffer pointed to by the data argument into the Scratch ring specified by address. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_ring_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Buffer to read from.
address	Address of the Scratch ring.
max_nn	Number of longwords to read from the buffer pointed to by val and writes it into the Scratch ring pointed to by address in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2 SRAM Operations

This section discusses the intrinsic functions that operate in the SRAM memory region. [Table 13](#) summarizes these functions.

Table 13. SRAM Operations Summary (Sheet 1 of 7)

Name (args)	Description
<pre>void sram_read[_D](__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads count longwords from SRAM at the specified address into the transfer register specified by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>
<pre>void sram_read[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads up to max_nn longwords from SRAM at the specified address into the transfer register specified by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_write[_D](__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes count longwords to SRAM at the specified address from the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_write[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes up to max_nn longwords to SRAM at the specified address from the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_set_bits[_D](__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Sets the bits in the specified mask in the longword at address in SRAM. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_set_bits[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Sets the bits in the specified mask in the longword at address in SRAM. The <code>ind</code> argument provides additional parameters and overrides. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_clear_bits[_D](__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Clears the bits in the specified mask in the longword at address in SRAM. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_clear_bits[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Clears the bits in the specified mask in the longword at address in SRAM. The <code>_D</code> version of this intrinsic must be used if the mask argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>

Table 13. SRAM Operations Summary (Continued) (Sheet 2 of 7)

Name (args)	Description
<pre>void sram_add[_D](__declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Increases the longword in SRAM at the specified address by the value specified in data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_add[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Increases the longword in SRAM at the specified address by the value specified in data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_incr(volatile void __declspec(sram) *address);</pre>	<p>Increases the longword in SRAM at the specified address by one.</p>
<pre>void sram_incr_ind(volatile void __declspec(sram) *address, sram_atomic_ind_t ind);</pre>	<p>Increases the longword in SRAM at the specified address by one. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_decr(volatile void __declspec(sram) *address);</pre>	<p>Decrements the longword in SRAM at the specified address by one.</p>
<pre>void sram_decr_ind(volatile void __declspec(sram) *address, sram_atomic_ind_t ind);</pre>	<p>Decrements the longword in SRAM at the specified address by one. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_swap[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into <code>val</code> the initial value of the longword at address. It then writes the value specified in data to the longword at address. The <code>_D</code> version of this intrinsic must be used if the <code>val</code> and data arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_swap[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into <code>val</code> the initial value of the longword at address. It then writes the value specified in data to the longword at address. The <code>_D</code> version of this intrinsic must be used if the <code>val</code> and data arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_test_and_set_bits[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into <code>val</code> the initial value of the longword at address in SRAM. It then sets the bits specified in <code>mask</code> in the longword at address in SRAM. The <code>_D</code> version of this intrinsic must be used if the <code>val</code> and data arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_test_and_set_bits[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into <code>val</code> the initial value of the longword at address in SRAM. It then sets the bits specified in <code>mask</code> in the longword at address in SRAM. The <code>ind</code> argument provides additional parameters and overrides. The <code>_D</code> version of this intrinsic must be used if the <code>val</code> and data arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>

Table 13. SRAM Operations Summary (Continued) (Sheet 3 of 7)

Name (args)	Description
<pre>void sram_test_and_clear_bits[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then clears the bits specified in mask in the longword at address in SRAM. The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_test_and_clear_bits[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *mask, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then clears the bits specified in mask in the longword at address in SRAM. The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>
<pre>void sram_test_and_add[_D](__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then increments the longword at address by the value specified by data. The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_test_and_add[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, __declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then increments the longword at address by the value specified by data. The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides additional parameters and overrides.</p>
<pre>void sram_test_and_incr[_D](__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then increments the longword at address by one. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>
<pre>void sram_test_and_incr[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then increments the longword at address by one. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>
<pre>void sram_test_and_decr[_D](__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then decrements the longword at address by one. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>
<pre>void sram_test_and_decr[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *val, volatile void __declspec(sram) *address, sram_atomic_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads into val the initial value of the longword at address in SRAM. It then decrements the longword at address by one. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>
<pre>void sram_csr_read[_D](__declspec([sram, dram]_read_reg) unsigned int *data, unsigned int address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads the value in the SRAM channel CSR specified by address into the transfer register specified by data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>

Table 13. SRAM Operations Summary (Continued) (Sheet 4 of 7)

Name (args)	Description
<pre>void sram_csr_read[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *data, unsigned int address, sram_csr_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Loads the value in the SRAM channel CSR specified by address into the transfer register specified by data. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_csr_write[_D](__declspec([sram, dram]_write_reg) unsigned int *data, unsigned int address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes the value from the transfer register specified by <code>data</code> to the SRAM channel CSR specified by address. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void sram_csr_write[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *data, unsigned int address, sram_csr_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes the value from the transfer register specified by <code>data</code> to the SRAM channel CSR specified by address. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_read_qdesc_head[_D](__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Issues a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM Queue Array as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Read Queue Descriptor). The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers.</p>
<pre>void sram_read_qdesc_head[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_read_qdesc_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Issues a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM Queue Array as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Read Queue Descriptor). The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_read_qdesc_tail[_D](__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Issues a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM Queue Array as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Read Queue Descriptor). The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers.</p>
<pre>void sram_read_qdesc_tail[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_read_qdesc_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Issues a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM Queue Array as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Read Queue Descriptor). The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void sram_read_qdesc_other(volatile void __declspec(sram) *address);</pre>	<p>Issues a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM Queue Array as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Read Queue Descriptor).</p>

Table 13. SRAM Operations Summary (Continued) (Sheet 5 of 7)

Name (args)	Description
<pre>void sram_read_qdesc_other_ind(volatile void __declspec(sram) *address, sram_read_qdesc_ind_t ind);</pre>	<p>Issues a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM Queue Array as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Read Queue Descriptor). The ind argument provides additional parameters and overrides.</p>
<pre>void sram_write_qdesc(volatile void __declspec(sram) *address);</pre>	<p>Issues a memory reference to an SRAM Channel to move data from the Queue Descriptor into SRAM as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Write Queue Descriptor).</p>
<pre>void sram_write_qdesc_count(volatile void __declspec(sram) *address);</pre>	<p>Issues a memory reference to an SRAM Channel to move data from the Queue Descriptor into SRAM as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(Write Queue Descriptor). This function differs from the sram_write_qdesc() only in the fields that are written to SRAM.</p>
<pre>void sram_enqueue(volatile void __declspec(sram) *address);</pre>	<p>Performs enqueue operations as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(enqueue).</p>
<pre>void sram_enqueue_ind(volatile void __declspec(sram) *address, sram_enqueue_ind_t ind);</pre>	<p>Performs enqueue operations as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(enqueue). The ind argument provides additional parameters and overrides.</p>
<pre>void sram_enqueue_tail(volatile void __declspec(sram) *address);</pre>	<p>Performs enqueue operations as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(enqueue).</p>
<pre>void sram_enqueue_tail_ind(volatile void __declspec(sram) *address, sram_enqueue_ind_t ind);</pre>	<p>Performs enqueue operations as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(enqueue). The ind argument provides additional parameters and overrides.</p>
<pre>void sram_dequeue[_D](__declspec([sram, dram] read_reg) void *data, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Performs a dequeue operation on the SRAM queue array specified by address, as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(dequeue). The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>
<pre>void sram_dequeue[_D]_ind(__declspec([sram, dram] read_reg) void *data, volatile void __declspec(sram) *address, sram_read_qdesc_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Performs a dequeue operation on the SRAM queue array specified by address, as described in the <i>IXP2400/IXP2800 Network Processor Programmer's Reference Manual</i> for the microcode instruction SRAM(dequeue). The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>

Table 13. SRAM Operations Summary (Continued) (Sheet 6 of 7)

Name (args)	Description
<pre>void sram_get_ring[_D](__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	Gets count longwords of data from the ring specified by address and returns it in data. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers.
<pre>void sram_get_ring[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_ring_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	Gets up to <code>max_nn</code> longwords of data from the SRAM ring specified by address and returns it in data. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.
<pre>void sram_put_ring[_D](__declspec([sram, dram]_read_reg) unsigned int *status, __declspec([sram, dram]_write_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	Puts count longwords of data from <code>val</code> into the ring specified by address. Returns the ring status through the status pointer. The <code>_D</code> version of this intrinsic must be used if the <code>status</code> and <code>data</code> arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.
<pre>void sram_put_ring[_D]_ind(__declspec([sram, dram]_read_reg) unsigned int *status, __declspec([sram, dram]_write_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_ring_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	Puts up to <code>max_nn</code> longwords of data from <code>data</code> into the SRAM ring specified by address. The <code>ind</code> argument provides additional parameters and overrides. Returns the ring status through the status pointer. The <code>_D</code> version of this intrinsic must be used if the <code>status</code> and <code>data</code> arguments are in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.
<pre>void sram_journal[_D](__declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	Puts count longwords from <code>data</code> into the journal ring specified by address. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.
<pre>void sram_journal[_D]_ind(__declspec([sram, dram]_write_reg) unsigned int *data, volatile void __declspec(sram) *address, unsigned int max_nn, sram_journal_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	Puts up to <code>max_nn</code> longwords from <code>data</code> into the journal ring specified by address. The <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.
<pre>void sram_fast_journal(volatile void __declspec(sram) *address);</pre>	Puts immediate data into the journal ring. Both the immediate data and the journal ring are specified by address.
<pre>void sram_fast_journal_ind(volatile void __declspec(sram) *address, sram_journal_ind_t ind);</pre>	Puts immediate data into the journal ring. Both the immediate data and the journal ring are specified by address. The <code>ind</code> argument provides additional parameters and overrides.
<pre>void sram_add_int(int data, volatile void __declspec(sram) *address);</pre>	(IXP28xx Rev. B and above only) This function increments the longword at address by the integer argument <code>data</code> . This integer must be an 11 bits or less in size, and will be sign-extended to 32 bits.
<pre>void sram_clear_bit_pos(unsigned int bit_pos, volatile void __declspec(sram) *address);</pre>	(IXP28xx Rev. B and above only) This function clears the bit specified in the integer <code>bit_pos</code> (0 = LSB, 31 = MSB) in the longword at address in SRAM.

Table 13. SRAM Operations Summary (Continued) (Sheet 7 of 7)

Name (args)	Description
<pre>void sram_set_bit_pos(unsigned int bit_pos, volatile void __declspec(sram) *address);</pre>	<p>(IXP28xx Rev. B and above only) This function sets the bit specified in the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM.</p>
<pre>void sram_swap_int[_D](__declspec([sram, dram]_read_reg) unsigned int *val, int data, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>(IXP28xx Rev. B and above only) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then writes the longword data to the longword at address in SRAM. data must be 11 bits or less, and will be sign-extended to 32 bits. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>
<pre>void sram_test_and_add_int[_D](__declspec([sram, dram]_read_reg) unsigned int *val, int data, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>(IXP28xx Rev. B and above only) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then increments the longword at address in SRAM by the longword data. data must be 11 bits or less, and will be sign-extended to 32 bits. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>
<pre>void sram_test_and_clear_bit_pos[_D](__declspec([sram, dram]_read_reg) unsigned int *val, unsigned int bit_pos, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>(IXP28xx Rev. B and above only) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then clears the bit specified with the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>
<pre>void sram_test_and_set_bit_pos[_D](__declspec([sram, dram]_read_reg) unsigned int *val, unsigned int bit_pos, volatile void __declspec(sram) *address, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>(IXP28xx Rev. B and above only) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then sets the bit specified with the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM. The _D version of this intrinsic must be used if the val argument is in DRAM transfers registers.</p>

4.3.3.2.1 `sram_read()`, `sram_read_D()`

Function Syntax:

```
void sram_read[_D] (
    __declspec([sram, dram] read_reg) void * data,
    volatile void __declspec(sram) * address,
    unsigned int count,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions read count longwords from SRAM at the specified address into the structure addressed by data. The `sram_read()` function puts the data into an SRAM register while the `sram_read_D()` function places the data in a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Address of data buffer to read into.
<code>address</code>	Address to read from.
<code>count</code>	Number of longwords to read/write in the range of 1 - 16.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.2 `sram_read_ind()`, `sram_read_D_ind()`

Function Syntax:

```
void sram_read[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read up to `max_nn` longwords from SRAM at the specified address and place the data into the structure addressed by `data`. The `sram_read()` function puts the data into an SRAM register while the `sram_read_D()` function places the data in a DRAM register. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_read_write_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `data` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Address of data buffer to read into.
<code>address</code>	Address to read from
<code>max_nn</code>	Number of longwords to read/write in the range of 1 - 16.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion

4.3.3.2.3 `sram_write()`, `sram_write_D()`

Function Syntax:

```
void sram_write[_D](
    __declspec([sram, dram]_write_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions write count longwords to SRAM at the specified address from the transfer register specified by the data argument. The `sram_write()` function transfers the data from an SRAM transfer register while the `sram_write_D()` function transfers the data from a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Address of data buffer to read from.
address	Address to write to.
count	Number of longwords to read and write in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.4 sram_write_ind(), sram_write_D_ind()

Function Syntax:

```
void sram_write[_D]_ind(
    __declspec([sram, dram]_write_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_read_write_ind_t ind
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions write up to max_nn longwords to SRAM at the specified address from the register specified by the data argument. The sram_write() function transfers the data from an SRAM transfer register while the sram_write_D() function transfers the data from a DRAM register. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_read_write_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Address of data buffer to read from.
address	Address to write to.
max_nn	Number of longwords to read and write in the range of 1 - 16.
ind	Indirect Word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion

4.3.3.2.5 `sram_set_bits()`, `sram_set_bits_D()`

Function Syntax:

```
void sram_set_bits[_D] (
    __declspec([sram, dram] write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions set the bits in the specified mask at the specified address in SRAM. The `sram_set_bits()` function obtains the mask from an SRAM register while the `sram_set_bits_D()` function obtains the mask from a DRAM register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

mask	Mask to set.
address	Address to write.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.



4.3.3.2.6 sram_set_bits_ind(), sram_set_bits_D_ind()

Function Syntax:

```
void sram_set_bits[_D]_ind(
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions set the bits in the specified mask at the specified address in SRAM. The ind argument provides additional parameters and overrides. The sram_set_bits_ind() function obtains the mask from an SRAM register while the sram_set_bits_D_ind() function obtains the mask from a DRAM register. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

mask	Mask to set.
address	Address to write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.7 `sram_clear_bits()`, `sram_clear_bits_D()`

Function Syntax:

```
void sram_clear_bits[_D] (
    __declspec([sram, dram] write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions clear the bits in the specified mask at the specified address in SRAM. The `sram_clear_bits()` function obtains the mask from an SRAM register while the `sram_clear_bits_D()` function obtains the mask from a DRAM register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

mask	Mask to clear.
address	Address to write.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.8 `sram_clear_bits_ind()`, `sram_clear_bits_D_ind()`

Function Syntax:

```
void sram_clear_bits[_D]_ind(  
    __declspec([sram, dram]_write_reg) unsigned int *mask,  
    volatile void __declspec(sram) *address,  
    sram_atomic_ind_t ind  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

These functions clear the bits in the specified mask at the specified address in SRAM. The `sram_clear_bits_ind()` function obtains the mask from an SRAM register while the `sram_clear_bits_D_ind()` function obtains the mask from a DRAM register. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_atomic_ind_t` data type. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used if the mask argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>mask</code>	Mask to clear.
<code>address</code>	Address to write.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.9 sram_add(), sram_add_D()

Function Syntax:

```
void sram_add[_D] (
    __declspec([sram, dram] write_reg) void *data,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions increment the longword at address by the value specified in the data argument. The sram_add() function obtains the value from an SRAM register while the sram_add_D() function obtains the value from a DRAM register. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Value to add.
address	Address to read from.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.10 sram_add_ind(), sram_add_D_ind()

Function Syntax:

```
void sram_add[_D]_ind(  
    __declspec([sram,dram]_write_reg) void *data,  
    volatile void __declspec(sram) *address,  
    sram_atomic_ind_t ind,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

These functions increment the longword at address by the value specified in the data argument. The sram_add_ind() function obtains the value from an SRAM register while the sram_add_D_ind() function obtains the value from a DRAM register. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Value to add.
address	Address to read from.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.11 sram_incr()**Function Syntax:**

```
void sram_incr(  
    volatile void __declspec(sram) *address)
```

Description:

This function increments the longword at address by one.

Arguments:

address	Address to read from.
---------	-----------------------



4.3.3.2.12 `sram_incr_ind()`

Function Syntax:

```
void sram_incr_ind(  
    volatile void __declspec(sram) *address,  
    sram_atomic_ind_t ind)
```

Description:

This function increments the longword at address by one. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type.

Arguments:

address	Address to read from.
ind	Indirect word.

4.3.3.2.13 sram_decr()**Function Syntax:**

```
void sram_decr(  
    volatile void __declspec(sram) *address)
```

Description:

This function decrements the longword at address in SRAM by one.

Arguments:

address	Address to read from.
---------	-----------------------



4.3.3.2.14 `sram_decr_ind()`

Function Syntax:

```
void sram_decr_ind(  
    volatile void __declspec(sram) *address,  
    sram_atomic_ind_t ind)
```

Description:

This function decrements the longword at address in SRAM by one. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_atomic_ind_t` data type.

Arguments:

address	Address to read from.
ind	Indirect word.

4.3.3.2.15 `sram_swap()`, `sram_swap_D()`

Function Syntax:

```
void sram_swap[_D](
    __declspec([sram, dram]_read_reg) unsigned int *val,
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

These functions swap the contents of the specified SRAM address with the contents of the transfer register specified by the data argument. Initially, they load the contents located at the specified address in SRAM into the transfer register specified by the val argument. They then write the contents of the data argument to the specified SRAM address. The `sram_swap()` function loads the value into an SRAM transfer register and writes the contents from an SRAM transfer register while the `sram_swap_D()` function loads and writes the values into and from DRAM transfer registers. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of this intrinsic must be used when both the `val` and `data` arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>val</code>	Contains the value from the specified SRAM address location before the swap.
<code>data</code>	Contents that are written to the specified SRAM address.
<code>address</code>	SRAM address to read from, then write to.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.16 sram_swap_ind(), sram_swap_D_ind()

Function Syntax:

```
void sram_swap_ind[_D] (
    __declspec([sram, dram]_read_reg) unsigned int *val,
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

These functions swap the contents of the specified SRAM address with the contents of the transfer register specified by the data argument. Initially, they load the contents located at the specified address in SRAM into the transfer register specified by the val argument. They then write the contents of the data argument to the specified SRAM address. The sram_swap() function loads the value into an SRAM transfer register and writes the contents from an SRAM transfer register while the sram_swap_D() function loads and writes the values into and from DRAM transfer registers. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Contains the initial value of the SRAM address location before the swap.
data	Contains the contents that are written to the SRAM address.
address	SRAM address to read from, then write to.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.17 `sram_test_and_set_bits()`, `sram_test_and_set_bits_D()`

Function Syntax:

```
void sram_test_and_set_bits[_D](
    __declspec([sram, dram]_read_reg) unsigned int *val,
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then sets the bits specified in the buffer mask in the longword at address in SRAM. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.18 `sram_test_and_set_bits_ind()`, `sram_test_and_set_bits_D_ind()`

Function Syntax:

```
void sram_test_and_set_bits[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then sets the bits specified in the buffer mask in the longword at address in SRAM. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the scratch_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used when both the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.19 sram_test_and_clear_bits(), sram_test_and_clear_bits_D()

Function Syntax:

```
void sram_test_and_clear_bits[_D] (
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then clears the bits specified in the buffer mask in the longword at address in SRAM. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.20 `sram_test_and_clear_bits_ind()`, `sram_test_and_clear_bits_D_ind()`

Function Syntax:

```
void sram_test_and_clear_bits[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *mask,
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then clears the bits specified in the buffer mask in the longword at address in sram. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the val and mask arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
mask	Mask to set/clear.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.21 `sram_test_and_add()`, `sram_test_and_add_D()`

Function Syntax:

```
void sram_test_and_add[_D] (
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then increments the longword at address in SRAM by the argument data. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to add.
address	Address to read/write.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2 , “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.22 sram_test_and_add_ind(), sram_test_and_add_D_ind()

Function Syntax:

```
void sram_test_and_add[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    __declspec([sram, dram]_write_reg) unsigned int *data,
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then increments the longword at address in SRAM by the argument data. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of this intrinsic must be used if the val and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value before write.
data	Value to add.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.23 `sram_test_and_incr()`, `sram_test_and_incr_D()`

Function Syntax:

```
void sram_test_and_incr[_D](
    __declspec([sram, dram] read_reg) unsigned int *val
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions load the initial value of the longword located at address in SRAM into the data buffer pointed to by val. They then increment the longword at address in SRAM by one. The `sram_test_and_incr()` function loads the value into an SRAM register while the `sram_test_and_incr_D()` function loads the value into a DRAM register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `val` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>val</code>	Value before write.
<code>address</code>	Address to read/write.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.24 sram_test_and_incr_ind(), sram_test_and_incr_D_ind()

Function Syntax:

```
void sram_test_and_incr[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions load the initial value of the longword located at address in SRAM into the data buffer pointed to by val. They then increment the longword at address in SRAM by one. The sram_test_and_incr_ind() function loads the value into an SRAM register while the sram_test_and_incr_D_ind() function loads the value into a DRAM register. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

val	Value before write.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.25 `sram_test_and_decr()`, `sram_test_and_decr_D()`

Function Syntax:

```
void sram_test_and_decr[_D] (
    __declspec([sram, dram] read_reg) unsigned int *val
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions load the initial value of the longword located at address in SRAM into the data buffer pointed to by val. They then decrement the longword at address in SRAM by one. The `sram_test_and_decr()` function loads the value into an SRAM register while the `sram_test_and_decr_D()` function loads the value into a DRAM register. The argument `sig_ptr` should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the `val` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>val</code>	Value before write.
<code>address</code>	Address to read/write.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.



4.3.3.2.26 sram_test_and_decr_ind(), sram_test_and_decr_D_ind()

Function Syntax:

```
void sram_test_and_decr[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *val
    volatile void __declspec(sram) *address,
    sram_atomic_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions load the initial value of the longword located at address in SRAM into the data buffer pointed to by val. They then decrement the longword at address in SRAM by one. The sram_test_and_decr_ind() function loads the value into an SRAM register while the sram_test_and_decr_D_ind() function loads the value into a DRAM register. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_atomic_ind_t data type. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

val	Value before write.
address	Address to read/write.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.27 `sram_csr_read()`, `sram_csr_read_D()`

Function Syntax:

```
void sram_csr_read[_D] (
    __declspec( [sram, dram]_read_reg) unsigned int * data,
    unsigned int address,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions load the value in the SRAM channel CSR specified by address into the buffer pointed to by data. The `sram_csr_read()` function loads the value into an SRAM register while the `sram_csr_read_D()` function loads the value into a DRAM register. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Address of the data buffer to load into.
<code>address</code>	Address of the SRAM channel CSR to read from.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.28 sram_csr_read_ind(), sram_csr_read_D_ind()

Function Syntax:

```
void sram_csr_read[_D]_ind(  
    __declspec([sram, dram]_read_reg) unsigned int * data,  
    unsigned int address,  
    sram_csr_read_write_ind_t ind,  
    sync_t sync,  
    SIGNAL* sig_ptr);
```

Description:

These functions load the value in the SRAM channel CSR specified by address into the buffer pointed to by data. The `sram_csr_read_ind()` function loads the value into an SRAM register while the `sram_csr_read_D_ind()` function loads the value into a DRAM register. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_csr_read_ind_t` data type. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Address of the data buffer to load into.
<code>address</code>	Address of the SRAM channel CSR to read from.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.29 `sram_csr_write()`, `sram_csr_write_D()`

Function Syntax:

```
void sram_csr_write[_D] (
    __declspec( [sram, dram] write_reg) unsigned int * data,
    unsigned int address,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions write to the SRAM channel CSR specified by address with the value specified by data. The `sram_csr_write()` function reads the data from an SRAM register while the `sram_csr_write_D` function reads the data from a DRAM register. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Address of data buffer to read from
address	Address of the SRAM channel CSR to write to.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.30 sram_csr_write_ind(), sram_csr_write_D_ind()

Function Syntax:

```
void sram_csr_write[_D]_ind(
    __declspec([sram, dram]_write_reg) unsigned int * data,
    unsigned int address,
    sram_csr_read_write_ind_t ind,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions write to the SRAM channel CSR specified by address from the buffer specified by data. The sram_csr_write_ind() function reads the data from an SRAM register while the sram_csr_write_D_ind function reads the data from a DRAM register. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_csr_read_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Address of data buffer to read from.
address	Address of the SRAM channel CSR to write to.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.31 `sram_read_qdesc_head()`, `sram_read_qdesc_head_D()`

Function Syntax:

```
void sram_read_qdesc_head[_D](
    __declspec([sram, dram] read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions issue a memory reference to an SRAM channel to read the queue descriptor into the SRAM Queue Array. (Refer to the SRAM(Read Queue Descriptor) instruction described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for more information.) The `sram_read_qdesc_head()` function reads the data from an SRAM register while the `sram_read_qdesc_head_D()` function reads the data from a DRAM register. Argument `count` is preferred to be a constant; otherwise the compiler generates an `indirect_ref` at the cost of performance. A constant `count` parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Transfer register where the <code>q_count</code> and optional data is read from.
address	Address of the SRAM channel CSR to write to.
count	Specifies the number of transfers (2 to 8) in increments of 4 byte words.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.32 `sram_read_qdesc_head_ind()`, `sram_read_qdesc_head_D_ind()`

Function Syntax:

```
void sram_read_qdesc_head[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_read_qdesc_ind_t ind,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions issue a memory reference to an SRAM channel to read the queue descriptor into the SRAM Queue Array. (Refer to the SRAM(Read Queue Descriptor) instruction described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for more information.) The `sram_read_qdesc_head_ind()` function reads the data from an SRAM register while the `sram_read_qdesc_head_D_ind()` function reads the data from a DRAM register. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_read_qdesc_ind_t` data type. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Transfer register where the <code>q_count</code> and optional data is read from.
<code>address</code>	Address of the SRAM channel CSR to write to.
<code>max_nn</code>	Specifies the number of transfers (2 to 8) in increments of 4 byte words.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.33 `sram_read_qdesc_tail()`, `sram_read_qdesc_tail_D()`

Function Syntax:

```
void sram_read_qdesc_tail[_D](
    __declspec([sram, dram] read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions issue a memory reference to an SRAM channel to read the queue descriptor into the SRAM Queue Array as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(Read Queue Descriptor). The `sram_read_qdesc_tail()` function reads the data from an SRAM register while the `sram_read_qdesc_head_D()` function reads the data from a DRAM register. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Transfer register where the <code>q_count</code> and optional data is read from.
address	Address of the SRAM channel CSR to write to.
count	Specifies the number of transfers (2 to 8) in increments of 4 byte words.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.34 `sram_read_qdesc_tail_ind()`, `sram_read_qdesc_tail_D_ind()`

Function Syntax:

```
void sram_read_qdesc_tail[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_read_qdesc_ind_t ind,
    sync_t sync,
    SIGNAL* sig_ptr);
```

Description:

These functions issue a memory reference to an SRAM channel to read the queue descriptor into the SRAM Queue Array as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(Read Queue Descriptor). The `sram_read_qdesc_tail_ind()` function reads the data from an SRAM register while the `sram_read_qdesc_head_D_ind()` function reads the data from a DRAM register. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_read_qdesc_ind_t` data type. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Transfer register where the <code>q_count</code> and optional data is returned.
<code>address</code>	Address of the SRAM channel CSR to write to.
<code>max_nn</code>	Specifies the number of transfers (2 to 8) in increments of 4 byte words.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.35 sram_read_qdesc_other()**Function Syntax:**

```
void sram_read_qdesc_other(  
    volatile void __declspec(sram) *address);
```

Description:

This function issues a memory reference to an SRAM channel to read the queue descriptor into the SRAM Queue Array as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(Read Queue Descriptor).

Arguments:

address Address of the SRAM channel CSR to write to.

4.3.3.2.36 `sram_read_qdesc_other_ind()`

Function Syntax:

```
void sram_read_qdesc_other_ind(  
    volatile void __declspec(sram) *address,  
    sram_read_qdesc_ind_t ind);
```

Description:

This function issues a memory reference to an SRAM channel to read the queue descriptor into the SRAM Queue Array as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(Read Queue Descriptor). The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_read_qdesc_ind_t` data type.

Arguments:

<code>address</code>	Address of the SRAM channel CSR to write to.
<code>ind</code>	Indirect word.

4.3.3.2.37 sram_write_qdesc()**Function Syntax:**

```
void sram_write_qdesc(  
    volatile void __declspec(sram) *address);
```

Description:

This function issues a memory reference to an SRAM channel to move data from the queue descriptor into the SRAM as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(Write Queue Descriptor).

Arguments:

address	Address in SRAM to write to.
---------	------------------------------



4.3.3.2.38 `sram_write_qdesc_count()`

Function Syntax:

```
void sram_write_qdesc_count(  
    volatile void __declspec(sram) *address);
```

Description:

This function issues a memory reference to an SRAM channel to move data from the queue descriptor into the SRAM as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(Write Queue Descriptor).

Arguments:

address	Address in SRAM to write to.
---------	------------------------------

4.3.3.2.39 sram_enqueue()**Function Syntax:**

```
void sram_enqueue(  
    volatile void __declspec(sram) *address);
```

Description:

This function performs enqueue operations as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(enqueue).

Arguments:

address	SRAM address.
---------	---------------

4.3.3.2.40 sram_enqueue_ind()

Function Syntax:

```
void sram_enqueue_ind(  
    volatile void __declspec(sram) *address,  
    sram_enqueue_ind_t ind);
```

Description:

This function performs enqueue operations as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(enqueue). The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_enqueue_ind_t data type.

Arguments:

address	SRAM address
ind	Indirect word.

4.3.3.2.41 sram_enqueue_tail()**Function Syntax:**

```
void sram_enqueue_tail(  
    volatile void __declspec(sram) *address);
```

Description:

This function performs enqueue operations as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(enqueue).

Arguments:

address	SRAM address.
---------	---------------

4.3.3.2.42 sram_enqueue_tail_ind()

Function Syntax:

```
void sram_enqueue_tail_ind(  
    volatile void __declspec(sram) *address,  
    sram_enqueue_ind_t ind);
```

Description:

This function performs enqueue operations as described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(enqueue). The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_enqueue_ind_t data type.

Arguments:

address	SRAM Address.
ind	Indirect word.

4.3.3.2.43 sram_dequeue(), sram_dequeue_D()

Function Syntax:

```
void sram_dequeue[_D] (
    __declspec([sram, dram] read_reg) void *data,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions perform an SRAM dequeue operation on the SRAM queue array specified by address. The sram_dequeue() function reads the data from an SRAM register while the sram_dequeue_D() function reads the data from a DRAM register. The SRAM dequeue operation is described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction SRAM(dequeue). The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Transfer register to read from.
address	Address of the SRAM queue array to write to.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.44 sram_dequeue_ind(), sram_dequeue_D_ind()

Function Syntax:

```
void sram_dequeue[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(sram) *address,
    sram_read_qdesc_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions perform an SRAM dequeue operation on the SRAM queue array specified by address. The sram_dequeue_ind() function reads the data from an SRAM register while the sram_dequeue_D_ind() function reads the data from a DRAM register. The SRAM dequeue operation is described in the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual* for the instruction sram(dequeue). The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_read_qdesc_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Transfer register to read from.
address	Address of the SRAM queue array to write to.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.45 `sram_get_ring()`, `sram_get_ring_D()`

Function Syntax:

```
void sram_get_ring[_D] (
    __declspec([sram, dram] read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions get count longwords of data from the ring specified by address and returns it in the buffer pointed to by data. The `sram_get_ring()` function returns the data to an SRAM register while the `sram_get_ring_D()` function returns the data to a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Data returned from the SRAM ring.
<code>address</code>	Address of the SRAM ring to read from.
<code>count</code>	Number of longwords to get in the range of 1 - 16.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.3.3.2.46 sram_get_ring_ind(), sram_get_ring_D_ind()

Function Syntax:

```
void sram_get_ring[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_ring_ind_t,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These function get up to max_nn longwords of data from the ring specified by address and returns it in the buffer pointed to by data. The sram_get_ring_ind() function returns the data to an SRAM register while the sram_get_ring_D_ind() function returns the data to a DRAM register. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_ring_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data returned from the SRAM ring.
address	Address of the SRAM ring to read from.
max_nn	Number of longwords to get in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.47 `sram_put_ring()`, `sram_put_ring_D()`

Function Syntax:

```
void sram_put_ring[_D] (
    __declspec([sram, dram]_read_reg) unsigned int *status,
    __declspec([sram, dram]_write_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function puts count longwords of data from the buffer pointed to by data into the ring specified by address. It returns the ring status through the status pointer. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the status and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

status	Returned 4-byte status word that indicates the current number of 4-byte words on the ring before the put operation and whether the put operation was successful. If the put operation is successful, bit 31 of the status word is set to 1. If not successful, bit 31 is set to 0.
data	Value to write to the specified SRAM ring.
address	Address of the SRAM ring.
count	Number of longwords to put in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to <code>__implicit_read()</code> on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.48 sram_put_ring_ind(), sram_put_ring_D_ind()

Function Syntax:

```
void sram_put_ring[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *status,
    __declspec([sram, dram]_write_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_ring_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function puts up to max_nn longwords of data from the buffer pointed to by data into the ring specified by address. It returns the ring status through the status pointer. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_ring_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the status and data arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

status	Returned 4-byte status word of the put operation that indicates the current number of 4-byte words on the ring before the put operation and whether the put operation was successful. If the put operation is successful, bit 31 of the status word is set to 1. If not successful, bit 31 is set to 0.
data	Value to write to the specified SRAM ring.
address	Address of the SRAM ring.
max_nn	Number of longwords to put in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.49 sram_journav(), sram_journal_D()

Function Syntax:

```
void sram_journal[_D] (
    __declspec([sram, dram] read_reg) unsigned int *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function puts count longwords of data from the buffer pointed by data into the journal ring specified by address. The count argument must be in the range of 1 through 16. Argument count specifies number of longwords if not overridden by ind, or the maximum number of longwords possibly overridden in ind. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Value to write into the SRAM journal ring.
address	Address of the SRAM journal.
count	Number of longwords to put into the journal in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.50 sram_journal_ind(), sram_journal_D_ind()

Function Syntax:

```
void sram_journal[_D]_ind(
    __declspec([sram, dram]_read_reg) unsigned int *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    sram_journal_ind_t,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function puts up to max_nn longwords of data from the buffer pointed to by data into the ring specified by address. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the sram_journal_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	Value to write into the SRAM journal ring.
address	Address of the SRAM journal.
max_nn	Number of longwords to put into the journal in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.3.3.2.51 sram_fast_journal()**Function Syntax:**

```
void sram_fast_journal(  
    volatile void __declspec(sram) *address);
```

Description:

This function puts immediate data into the journal ring. Both the immediate data and the journal ring are specified by address.

Arguments:

address	Address of the immediate data and the journal ring.
---------	---



4.3.3.2.52 `sram_fast_journal_ind()`

Function Syntax:

```
void sram_fast_journal_ind(  
    volatile void __declspec(sram) *address,  
    sram_journal_ind_t ind);
```

Description:

This function puts immediate data into the journal ring. Both the immediate data and the journal ring are specified by address. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `sram_journal_ind_t` data type.

Arguments:

<code>address</code>	Address of the immediate data and the journal ring.
<code>ind</code>	Indirect word.

4.3.3.2.53 sram_add_int()**Function Syntax:**

```
void sram_add_int(  
    int data,  
    volatile void __declspec(sram) *address);
```

Description:

(IXP28xx Rev. B and above only) This function increments the longword at address by the integer argument data. This integer must be 11 bits or less in size, and will be sign-extended to 32 bits.

Arguments:

data	11-bit sign-extended data to add.
address	The memory location to modify.



4.3.3.2.54 sram_clear_bit_pos()

Function Syntax:

```
void sram_clear_bit_pos(  
    unsigned int bit_pos,  
    volatile void __declspec(sram) *address);
```

Description:

(IXP28xx Rev. B and above only) This function clears the bit specified in the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM.

Arguments:

bit_pos	The bit to clear. Must be in the range 0:31.
address	The memory location to modify.

4.3.3.2.55 sram_set_bit_pos()**Function Syntax:**

```
void sram_set_bit_pos(  
    unsigned int bit_pos,  
    volatile void __declspec(sram) *address);
```

Description:

(IXP28xx Rev. B and above only) This function sets the bit specified in the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM.

Arguments:

bit_pos	The bit to set. Must be in the range 0:31.
address	The memory location to modify.

4.3.3.2.56 sram_swap_int(), sram_swap_int_D()

Function Syntax:

```
void sram_swap_int[_D] (
    __declspec([sram, dram]_read_reg) unsigned int *val,
    int data,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

(IXP28xx Rev. B and above only) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then writes the longword data to the longword at address in SRAM. data must be 11 bits or less, and will be sign-extended to 32 bits. The sram_swap_int() function loads the initial value to SRAM transfer register while the sram_swap_int_D function loads the initial value to a DRAM transfer register. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	The pre-modified value.
data	The 11-bit sign-extended data to swap.
address	The memory location to modify.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be sig_done. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	The signal to raise upon completion.

4.3.3.2.57 `sram_test_and_add_int()`, `sram_test_and_add_int_D()`

Function Syntax:

```
void sram_test_and_add_int [D] (
    __declspec( [sram, dram]_read_reg) unsigned int *val,
    int data,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

(IXP28xx Rev. B and above only.) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then increments the longword at address in SRAM by the longword data. data must be 11 bits or less, and will be sign-extended to 32 bits. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The D version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	The pre-modified value.
data	The 11-bit sign-extended data to swap.
address	The memory location to modify.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be sig_done. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	The signal to raise upon completion.

4.3.3.2.58 `sram_test_and_clear_bit_pos()`, `sram_test_and_clear_bit_pos_D()`

Function Syntax:

```
void sram_test_and_clear_bit_pos[_D](
    __declspec([sram, dram]_read_reg) unsigned int *val,
    unsigned int bit_pos,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

(IXP28xx Rev. B and above only) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then clears the bit specified with the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	The pre-modified value.
bit_pos	The bit to clear. Must be in the range 0:31
address	The memory location to modify.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be sig_done. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to ¹ for details and an example.
sig_ptr	The signal to raise upon completion.

1.

4.3.3.2.59 sram_test_and_set_bit_pos(), sram_test_and_set_bit_pos_D()

Function Syntax:

```
void sram_test_and_set_bit_pos[_D] (
    __declspec([sram, dram]_read_reg) unsigned int *val,
    unsigned int bit_pos,
    volatile void __declspec(sram) *address,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

(IXP28xx Rev. B and above only.) This function loads the initial value of the longword located at address in SRAM into the data buffer pointed to by val. It then sets the bit specified with the integer bit_pos (0 = LSB, 31 = MSB) in the longword at address in SRAM. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the val argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

val	The pre-modified value.
bit_pos	The bit to set. Must be in the range 0:31
address	The memory location to modify.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be sig_done. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	The signal to raise upon completion.

4.3.3.3 DRAM Operations

This section discusses DRAM operations. [Table 14](#) provides a summary of these operations.

Table 14. DRAM Operations Summary

Name (args)	Description
<pre>void dram_read[_S](__declspec([dram, sram] read_reg) void *data, volatile void __declspec(dram) *address, unsigned int count, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Reads count quadwords from DRAM at the specified address into the transfer register addressed by data. The <code>_S</code> version of this intrinsic must be used if the data argument is in SRAM transfers registers.</p>
<pre>void dram_read[_S]_ind(__declspec([dram, sram] read_reg) void *data, volatile void __declspec(dram) *address, unsigned int max_nn, dram_read_write_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Reads up to max_nn quadwords from DRAM at the specified address into the structure addressed by data. The <code>_S</code> version of this intrinsic must be used if the data argument is in SRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void dram_write[_S](__declspec([dram, sram] write_reg) void *data, volatile void __declspec(dram) *address, unsigned int count, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Writes count quadwords to DRAM at the specified address from the structure addressed by data. The <code>_S</code> version of this intrinsic must be used if the data argument is in SRAM transfers registers. The <code>_S</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void dram_write[_S]_ind(__declspec([dram, sram] write_reg) void *data, volatile void __declspec(dram) *address, unsigned int max_nn, dram_read_write_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Writes up to max_nn quadwords to DRAM at the specified address from the structure addressed by data. The <code>_S</code> version of this intrinsic must be used if the data argument is in SRAM transfers registers. The <code>_S</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void dram_rbuf_read_ind(volatile void __declspec(dram) *address, unsigned int max_nn, dram_rbuf_tbuf_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Moves max_nn quadwords of data directly from the receive FIFO to DRAM at address. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void dram_tbuf_write_ind(volatile void __declspec(dram) *address, unsigned int max_nn, dram_rbuf_tbuf_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Moves max_nn quadwords of data directly from DRAM at address to the transmit FIFO. The <code>ind</code> argument provides additional parameters and overrides.</p>

4.3.3.3.1 dram_read(), dram_read_S()

Function Syntax:

```
void dram_read[_S] (
    __declspec( [dram, sram] read_reg) void *data,
    volatile void __declspec(dram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

These functions read count quadwords of data from DRAM at the specified address and loads the data into the structure addressed by data. The dram_read() function loads the data in a DRAM register while the dram_read_S() function loads the data in an SRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr should be the address of a user signal variable passed by direct reference.

Note: The _S version of the intrinsic must be used if the data argument is in SRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Address of the register to load into.
address	Address of the location in DRAM to read from.
count	Number of quadwords to read in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal pair to use.

4.3.3.3.2 dram_read_ind(), dram_read_S_ind()

Function Syntax:

```
void dram_read[_S]_ind(
    __declspec([dram, sram]_read_reg) void *data,
    volatile void __declspec(dram) *address,
    unsigned int max_nn,
    dram_read_write_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

These functions read count quadwords of data from DRAM at the specified address and loads the data into the structure addressed by data. The `dram_read_ind()` function loads the data in a DRAM register while the `dram_read_S_ind()` function loads the data in an SRAM register. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of quadwords to be transferred while the `ind` argument specifies the exact number of quadwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of quadwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `dram_read_write_ind_t` data type. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_S` version of the intrinsic must be used if the data argument is in SRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Address of the register to load the data into.
<code>address</code>	Address of the DRAM location to read from.
<code>ind</code>	Indirect word.
<code>max_nn</code>	Number of quadwords to read in the range of 1 - 16.
<code>sync</code>	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Pointer to signal pair to use.

4.3.3.3.3 dram_write(), dram_write_S()

Function Syntax:

```
void dram_write[_S](
    __declspec([dram, sram] write_reg) void *data,
    volatile void __declspec(dram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function writes count quadwords to DRAM at the specified address from the structure addressed by data. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The “_S” version of the intrinsic must be used if the data argument is in SRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Address of data buffer to read from
address	Address of the DRAM location to write to.
count	Number of quadwords to write in the range of 1 - 16.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal pair to use.

4.3.3.3.4 dram_write_ind(), dram_write_S_ind()

Function Syntax:

```
void dram_write[_S]_ind(
    __declspec([dram, sram]_write_reg) void *data,
    volatile void __declspec(dram) *address,
    unsigned int max_nn,
    dram_read_write_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function writes up to max_nn quadwords to DRAM at the specified address from the structure addressed by data. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of quadwords to be transferred while the ind argument specifies the exact number of quadwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of quadwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the dram_read_write_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _S version of the intrinsic must be used if the data argument is in SRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Address of data buffer to read from.
address	Address of the DRAM location to write to.
max_nn	The number of quadwords to write in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal pair to use.

4.3.3.3.5 dram_rbuf_read_ind()

Function Syntax:

```
void dram_rbuf_read_ind(
    volatile void __declspec(dram) *address,
    unsigned int max_nn,
    dram_rbuf_tbuf_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function moves max_nn quadwords of data directly from the receive FIFO to DRAM at address. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of quadwords to be transferred while the ind argument specifies the exact number of quadwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of quadwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the dram_rbuf_tbuf_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Arguments:

address	Address of the DRAM location to write to.
max_nn	The number of quadwords to write in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use.
sig_ptr	Pointer to signal pair to use.

4.3.3.3.6 dram_tbuf_write_ind()

Function Syntax:

```
void dram_tbuf_write_ind(  
    volatile void __declspec(dram) *address,  
    unsigned int max_nn,  
    dram_rbuf_tbuf_ind_t ind,  
    sync_t sync,  
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function moves max_nn quadwords of data directly from DRAM at address to the transmit FIFO. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of quadwords to be transferred while the ind argument specifies the exact number of quadwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of quadwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the dram_rbuf_tbuf_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Arguments:

address	Address of the DRAM location to read from.
max_nn	The number of quadwords to write in the range of 1 -1 6.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant.
sig_ptr	Pointer to signal pair to use.

4.3.3.4 MSF Operations

This section discusses MSF functions. [Table 15](#) summarizes these operations.

Table 15. MSF Operations Summary (Sheet 1 of 2)

Name (args)	Description
<pre>void msf_read[_D](__declspec([sram, dram]_read_reg) void *data, volatile void *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads count longwords from the MSF (media switch fabric) at the specified address into the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>
<pre>void msf_read[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void *address, unsigned int max_nn, msf_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads up to max_nn longwords from the MSF (media switch fabric) at the specified address into the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void msf_read64[_D](__declspec([sram, dram]_read_reg) void *data, volatile void *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads count quadwords from the MSF (media switch fabric) at the specified address into the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>
<pre>void msf_read64[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void *address, unsigned int max_nn, msf_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads up to max_nn quadwords from the MSF (media switch fabric) at the specified address into the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void msf_write[_D](__declspec([sram, dram]_write_reg) void *data, volatile void *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes count longwords from data to the MSF (media switch fabric) at the specified address. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void msf_write[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void *address, unsigned int max_nn, msf_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes up to max_nn longwords from data to the MSF (media switch fabric) at the specified address. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>

Table 15. MSF Operations Summary (Continued) (Sheet 2 of 2)

Name (args)	Description
<pre>void msf_write64[_D](__declspec([sram, dram]_write_reg) void *data, volatile void *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes count quadwords from data to the MSF (media switch fabric) at the specified address. he <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void msf_write64[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void *address, unsigned int max_nn, msf_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes up to <code>max_nn</code> quadwords from data to the MSF (media switch fabric) at the specified address. he <code>_D</code> version of this intrinsic must be used if the <code>data</code> argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void msf_fast write(volatile void *address);</pre>	<p>Writes immediate data to MSF (media switch fabric). Both the immediate data and the journal ring are specified by address.</p>

4.3.3.4.1 msf_read(), msf_read_D()

Function Syntax:

```
void mfs_read[_D] (
    __declspec( [sram, dram]_read_reg) void *data,
    volatile void *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read count longwords of data from the MSF (media switch fabric) address specified by address and loads the data into the structure addressed by data. The `mfs_read()` function loads the data into an SRAM register while the `msf_read_D()` function loads the data into a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. The argument `sig_ptr`, should be the address of a user signal variable passed by direct reference.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	Data buffer to read into.
<code>address</code>	Address of the location in the MSF to read from.
<code>count</code>	Number of longwords to read in the range of 1 - 16.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Pointer to signal to use.



4.3.3.4.2 msf_read_ind(), msf_read_D_ind()

Function Syntax:

```
void msf_read[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void *address,
    unsigned int max_nn,
    msf_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read count longwords of data from the MSF (media switch fabric) address specified by address and loads the data into the structure addressed by data. The mfs_read() function loads the data into an SRAM register while the msf_read_D() function loads the data into a DRAM register. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the msf_read_write_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data buffer to read into.
address	Address of the msf location to read from.
max_nn	Number of longwords to read in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to ¹ for details and an example.
sig_ptr	Pointer to signal to use.

1.

4.3.3.4.3 msf_read64(), msf_read64_D()

Function Syntax:

```
void mfs_read64 [_D] (
    __declspec( [sram, dram] _read_reg) void *data,
    volatile void *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read count quadwords of data from the MSF (media switch fabric) address specified by address and loads the data into the structure addressed by data. The mfs_read() function loads the data into an SRAM register while the msf_read_D() function loads the data into a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data buffer to read into.
address	Address of the location in the MSF to read from.
count	Number of quadwords to read in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.4.4 msf_read64_ind(), msf_read64_D_ind()

Function Syntax:

```
void msf_read64[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void *address,
    unsigned int max_nn,
    msf_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read count quadwords of data from the MSF (media switch fabric) address specified by address and loads the data into the structure addressed by data. The mfs_read() function loads the data into an SRAM register while the msf_read_D() function loads the data into a DRAM register. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the msf_read_write_ind_t data type. The argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data buffer to read into.
address	Address of the MSF location to read from.
max_nn	Number of quadwords to read in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.4.5 msf_write(), msf_write_D()

Function Syntax:

```
void msf_write[_D](
    __declspec([sram, dram]_write_reg) void *data,
    volatile void *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes count longwords from the structure addressed by data into the MSF (media switch fabric) address specified by address. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr identifies your signal variable to use for signalling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Data buffer to read from.
address	Address of the location in the MSF to write to.
count	Number of longwords to write in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.4.6 `msf_write_ind()`, `msf_write_D_ind()`

Function Syntax:

```
void msf_write[_D]_ind(
    __declspec([sram, dram]_write_reg) void *data,
    volatile void *address,
    unsigned int max_nn,
    msf_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes up to `max_nn` longwords from the structure addressed by `data` into the MSF (media switch fabric) address specified by `address`. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `msf_read_write_ind_t` data type. The argument `sig_ptr` identifies your signal variable to use for signaling event completion.

Note: The `_D` version of the intrinsic must be used if the `data` argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Data buffer to read from.
<code>address</code>	Address of the MSF location to write to.
<code>max_nn</code>	Number of longwords to read in the range of 1 -16.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Pointer to signal to use.

4.3.3.4.7 `msf_write64()`, `msf_write64_D()`

Function Syntax:

```
void mfs_write64[_D] (
    __declspec([sram, dram]_write_reg) void *data,
    volatile void *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes count quadwords from the structure addressed by data into the MSF (media switch fabric) address specified by address. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Data buffer to read from.
address	Address of the location in the MSF to write to.
count	Number of quadwords to read in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.4.8 `msf_write64_ind()`, `msf_write64_D_ind()`

Function Syntax:

```
void msf_write64[_D]_ind(
    __declspec([sram, dram]_write_reg) void *data,
    volatile void *address,
    unsigned int max_nn,
    msf_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes up to `max_nn` quadwords from the structure specified by `data` into the MSF (media switch fabric) address specified by `address`. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `msf_read_write_ind_t` data type. The argument `sig_ptr` identifies your signal variable to use for signaling event completion.

Note: The `_D` version of the intrinsic must be used if the `data` argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Data buffer to read from.
<code>address</code>	Address of the MSF location to write to.
<code>max_nn</code>	Number of quadwords to read in the range of 1 - 16.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Pointer to signal to use.

4.3.3.4.9 msf_fast_write()**Function Syntax:**

```
void mfs_fast_write(  
    volatile void *address);
```

Description:

This function writes immediate data to the MSF (media switch fabric). Both the immediate data and the journal ring are specified by address.

Arguments:

address	Address of the location in the MSF to write to.
---------	---

4.3.3.5 PCI Operations

This section discusses PCI operations. [Table 16](#) summarizes these operations.

Table 16. PCI Operations Summary

Name (args)	Description
<pre>void pci_read[_D](__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads count longwords from the specified PCI address into the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers.</p>
<pre>void pci_read[_D]_ind(__declspec([sram, dram]_read_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, pci_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads up to max_nn longwords from the specified PCI address into the structure addressed by data. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>ind</code> argument provides additional parameters and overrides.</p>
<pre>void pci_write[_D](__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(sram) *address, unsigned int count, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes count longwords from data to the specified PCI address. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void pci_write[_D]_ind(__declspec([sram, dram]_write_reg) void *data, volatile void __declspec(sram) *address, unsigned int max_nn, pci_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes up to max_nn longwords from data to the specified PCI address. The <code>_D</code> version of this intrinsic must be used if the data argument is in DRAM transfers registers. The <code>_D</code> version is only available on IXP28XX hardware in 8-context mode. The <code>ind</code> argument provides additional parameters and overrides.</p>

4.3.3.5.1 pci_read(), pci_read_D()

Function Syntax:

```
void pci_read[_D] (
    __declspec([sram, dram] read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read count longwords of data from the PCI address specified by the address argument and load the data into the register specified by the data argument. The pci_read() function loads the data into an SRAM register while the pci_read_D() function loads the data into a DRAM register. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Register to read into.
address	Address of the location in PCI to read from.
count	Number of longwords to read in the range of 1 - 16.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.



4.3.3.5.2 pci_read_ind(), pci_read_D_ind()

Function Syntax:

```
void pci_read[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    pci_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read up to max_nn longwords from the PCI address specified by the address argument into the register specified by the data argument. The pci_read_ind() function loads the data into an SRAM register while the pci_read_D_ind() function loads the data into a DRAM register. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the pci_read_write_ind_t data type. The argument sig_ptr, should be the address of a user signal variable passed by direct reference.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data buffer to read into.
address	Address of the PCI location to read from.
max_nn	Number of longwords to read in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to ¹ for details and an example.
sig_ptr	Pointer to signal to use.

1.

4.3.3.5.3 pci_write(), pci_write_D()

Function Syntax:

```
void pci_write[_D] (
    __declspec([sram, dram] write_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int count,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes count longwords from the structure addressed by data into the PCI address specified by address. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. The argument sig_ptr identifies your signal variable to use for signalling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Data buffer to read from.
address	Address of the location in PCI to write to.
count	Number of longwords to write in the range of 1 -16.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.5.4 pci_write_ind(), pci_write_D_ind()

Function Syntax:

```
void pci_write[_D]_ind(
    __declspec([sram, dram]_write_reg) void *data,
    volatile void __declspec(sram) *address,
    unsigned int max_nn,
    pci_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes up to max_nn longwords from the structure addressed by data into the PCI address specified by address. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the pci_read_write_ind_t data type. The argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Data buffer to read from.
address	Address of the PCI location to write to.
max_nn	Number of longwords to read in the range of 1 - 16.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.6 Reflector Operations

This section discusses Reflector operations. [Table 17](#) summarizes these operations. Refer to [Section 4.5.2](#) for information on CAP operations with reflect.

Table 17. Reflector Operations Summary

Name (args)	Description
<pre>void reflect_read[_D](__declspec(sram_read_reg) void *data, unsigned int remote_ME, volatile __declspec(remote [sram, dram]_write_reg) void* remote_xfer, unsigned int remote_ctx, unsigned int count, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads count longwords from the remote transfer register specified by the arguments remote_ME, remote_xfer, and remote_ctx, into the structure addressed by data. The _D version of this intrinsic must be used if the remote_xfer argument is in DRAM transfers registers.</p>
<pre>void reflect_read[_D]_ind(__declspec(sram_read_reg) void *data, unsigned int remote_ME, volatile __declspec(remote [sram, dram]_write_reg) void* remote_xfer, unsigned int remote_ctx, unsigned int max_nn, reflect_read_write_ind_t ind, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Reads up to max_nn longwords from the remote transfer register specified by the arguments remote_ME, remote_xfer, and remote_ctx, into the structure addressed by data. The _D version of this intrinsic must be used if the remote_xfer argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>
<pre>void reflect_write[_D](__declspec(sram_write_reg) void *data, unsigned int remote_ME, volatile __declspec(remote [sram, dram]_read_reg) void* remote_xfer, unsigned int remote_ctx, unsigned int count, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes count longwords from the structure addressed by data to the remote transfer register specified by the arguments remote_ME, remote_xfer, and remote_ctx. The _D version of this intrinsic must be used if the remote_xfer argument is in DRAM transfers registers.</p>
<pre>void reflect_write[_D]_ind(__declspec(sram_write_reg) void *data, unsigned int remote_ME, volatile __declspec(remote [sram, dram]_read_reg) void* remote_xfer, unsigned int remote_ctx, unsigned int max_nn, reflect_read_write_ind_t ind, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);</pre>	<p>Writes up to max_nn longwords from the structure addressed by data to the remote transfer register specified by arguments remote_ME, remote_xfer, and remote_ctx. The _D version of this intrinsic must be used if the remote_xfer argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.</p>

4.3.3.6.1 reflect_read(), reflect_read_D()

Function Syntax:

```
void reflect_read[_D] (
    __declspec(sram_read_reg) void *data,
    unsigned int remote_ME,
    volatile __declspec(remote [sram, dram]_write_reg) void*
    remote_xfer,
    unsigned int remote_ctx,
    unsigned int count,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read count longwords of data from the remote transfer registers specified by the arguments `remote_ME`, `remote_xfer`, and `remote_ctx`, into the SRAM transfer register specified by the `data` argument. The `reflect_read()` function reads the data from a remote SRAM register while the `reflect_read_D()` function reads the data from a remote DRAM register. Arguments `remote_ME`, `remote_ctx`, `reflect_sig`, and `sync` must be constants. The count argument must be in the range of 1 through 16 and is preferred to be a constant; otherwise, the compiler generates an `indirect_ref` at the cost of performance. A constant count parameter that is larger than 8 also results in `indirect_ref` being generated. Argument `sig_ptr` identifies your signal variable to use for signaling event completion.

Note: The `_D` version of the intrinsic must be used if the `remote_xfer` argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

<code>data</code>	SRAM Transfer register to read into.
<code>remote_ME</code>	Remote ME to read from.
<code>remote_xfer</code>	Remote transfer register to read from.
<code>remote_ctx</code>	Remote ctx to read from.
<code>count</code>	Number of longwords to read in the range of 1 - 16.
<code>reflect_sig</code>	Reflect signal delivery type.
<code>sync</code>	Type of synchronization to use. If 'sig_done' synchronization is used, a call to the <code>__implicit_read()</code> intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example. Regardless of the type of synchronization, a call to <code>__implicit_write()</code> in the other ME's code may be required to let the compiler know the start of the lifetime of the transfer register data.
<code>sig_ptr</code>	Pointer to signal to use.

4.3.3.6.2 reflect_read_ind(), reflect_read_D_ind()

Function Syntax:

```
void reflect_read[_D]_ind(
    __declspec(sram_read_reg) void *data,
    unsigned int remote_ME,
    volatile __declspec(remote [sram, dram]_write_reg) void*
    remote_xfer,
    unsigned int remote_ctx,
    unsigned int max_nn,
    reflect_read_write_ind_t ind,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read up to max_nn longwords of data from the remote transfer register specified by the remote_ME, remote_xfer, and remote_ctx arguments into the SRAM transfer register specified by data. The reflect_read_ind() function reads the data from a remote SRAM register while the reflect_read_D_ind() function reads the data from a remote DRAM register. Arguments remote_ME, remote_ctx, reflect_sig, and sync must be constants. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides overrides and additional parameters. There are restrictions on the value specified in the override as noted in the description of the reflect_read_write_ind_t datatype. Argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the remote_xfer argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Transfer register to read into.
remote_ME	Remote ME to read from.
remote_xfer	Remote transfer register to read from.
remote_ctx	Remote ctx to read from.
max_nn	Number of longwords to read in the range of 1 - 16.
ind	Indirect word.
reflect_sig	Reflect signal delivery type.
sync	Type of synchronization to use. If 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example. Regardless of the type of synchronization, a call to __implicit_write() in the other ME's code may be required to let the compiler know the start of the lifetime of the transfer register data.
sig_ptr	Pointer to signal to use.

4.3.3.6.3 reflect_write(), reflect_write_D()

Function Syntax:

```
void reflect_write[_D] (
    __declspec(sram_write_reg) void *data,
    unsigned int remote_ME,
    volatile __declspec(remote [sram, dram]_read_reg) void*
    remote_xfer,
    unsigned int remote_ctx,
    unsigned int count,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions write count longwords of data from the SRAM transfer register specified by the data argument to the remote transfer register specified by arguments remote_ME, remote_xfer, and remote_ctx. The reflect_write() function writes the data to a remote SRAM transfer register while the reflect_write_D() function writes the data to a remote DRAM transfer register. The count argument must be in the range of 1 through 16. Arguments remote_ME, remote_ctx, reflect_sig, and sync must be constants. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref at the cost of performance. A constant count parameter that is larger than 8 also results in indirect_ref being generated. Argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the remote_xfer argument is in DRAM transfer registers.

Arguments:

data	SRAM transfer register to read from.
remote_ME	Remote ME to write to.
remote_xfer	Remote transfer register to write to.
remote_ctx	Remote ctx to write to.
count	Number of longwords to write in the range of 1 - 16.
reflect_sig	Reflect signal delivery type.
sync	Type of synchronization to use. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Regardless of the type of synchronization, a call to __implicit_read() in the other ME's code may be required to prolong the lifetime of the transfer register data. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.3.6.4 `reflect_write_ind()`, `reflect_write_D_ind()`

Function Syntax:

```
void reflect_write[_D]_ind(
    __declspec(sram_write_reg) void *data,
    unsigned int remote_ME,
    volatile __declspec(remote [sram, dram]_read_reg) void*
    remote_xfer,
    unsigned int remote_ctx,
    unsigned int max_nn,
    reflect_read_write_ind_t ind,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions write max_nn longwords of data from the SRAM transfer register specified by the data argument to the remote transfer register specified by the arguments remote_ME, remote_xfer, and remote_ctx. The reflect_write_ind() function writes the data to a remote SRAM transfer register while the reflect_write_D_ind() function writes the data to a remote DRAM transfer register. Arguments remote_ME, remote_ctx, and reflect_sig must be constants. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides overrides and additional parameters. There are restrictions on the value specified in the override as noted in the description of the reflect_read_write_ind_t datatype. Argument sig_ptr identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the remote_xfer argument is in DRAM transfer registers.

Arguments:

data	Transfer register to read from.
remote_ME	Remote ME to write to.
remote_xfer	Remote transfer register to write to.
remote_ctx	Remote ctx to write to.
max_nn	Number of longwords to write in the range of 1 - 16.
ind	Indirect word.
reflect_sig	Reflect signal delivery type.
sync	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Regardless of the type of synchronization, a call to __implicit_read() in the other ME's code may be required to prolong the lifetime of the transfer register data. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Pointer to signal to use.

4.3.4 Limitations on Some I/O Functions

Some hardware instructions only take one operand for both source and destination transfer register operand, but the intrinsic functions provide two separate parameters. These functions are:

- `scratch_swap`
- `scratch_swap_ind`
- `scratch_test_and_clear_bits`
- `scratch_test_and_clear_bits_ind`
- `scratch_test_and_set_bits`
- `scratch_test_and_set_bits_ind`
- `scratch_test_and_add`
- `scratch_test_and_add_ind`
- `scratch_test_and_sub`
- `scratch_test_and_sub_ind`
- `sram_swap`
- `sram_swap_ind`
- `sram_test_and_clear_bits`
- `sram_test_and_clear_bits_ind`
- `sram_test_and_set_bits`
- `sram_test_and_set_bits_ind`
- `sram_test_and_add`
- `sram_test_and_add_ind`
- `sram_put_ring`
- `sram_put_ring_ind`

The compiler internally maps both the read and write transfer registers to the same physical register number. As a result, certain usage of these functions is not legal. For example:

```

__declspec(sram_read_reg) int r1[2], r2, r3;
__declspec(sram_write_reg) int w1, w2, w3;
__declspec(sram) int p;
SIGNAL s1;

...
sram_swap(&r1[0], &w1, &p, sig_done, &s1); // r1[0] and w1 are mapped to the
                                           //same physical register number
__wait_for_all(&s1);

sram_swap(&r1[1], &w1, &p, sig_done, &s1); // Error: r1[1] and w1 can not be
                                           //mapped to the same physical
                                           //register number
__wait_for_all(&s1);

...

```

Another case:

```
sram_swap(&r2, &w1, &p, ctx_swap, &s1); // r2 and w1 are mapped to the same
                                           // physical register number
__wait_for_all(&s1);
sram_swap(&r3, &w1, &p, ctx_swap, &s1); // r3 and w1 are mapped to the same
                                           // physical register number
__wait_for_all(&s1);
__asm alu[X, --, B, r2]// Error: the second sram_swap corrupted
                       // r2 because r2 and r3 are mapped to the same
                       // physical register.
```

4.4 Synchronization Functions

This section describes the data types and functions used for synchronization.

4.4.1 Synchronization Data Types

4.4.1.1 `signal_t`

This enumeration type defines the special type of signals that can be tested by the `ctx_wait` functions. The values are:

```
enum signal_t{
    kill,
    voluntary,
    bpt,
    no_load
}
```

The `kill` signal puts the context into the Sleep state and does not return to the Ready state.

The `voluntary` signal puts the context into the Sleep state. The context is put back into the Ready state in one cycle since the Voluntary Event Signal is always set.

The `bpt` (breakpoint) stops all contexts, notifies the Intel Xscale® core processor, and puts the current context into the Sleep state. It also sets the `CTX_Enable[Breakpoint]` bit. This value is typically used for debugging purposes. For more information the use of this value, refer to the `ctx_arb` instruction in the *IXP2400/IXP2800 Network Processor Microcode Programmer's Reference Manual*.

The `no_load` value instructs `ctx_swap` to use the last signal mask written into local CSRs.

4.4.1.2 `SIGNAL_MASK`

The `SIGNAL_MASK` data type is used for masks specifying one or more signal registers. It is typedef'ed as follows

```
typedef int SIGNAL_MASK;
```

4.4.1.3 `inp_state_t`

This enumeration defines the range of state values that can be tested with the `inp_state_test()` intrinsic. The values are:

```
enum inp_state_t{
    inp_state_nn_empty,
    inp_state_nn_full,
    inp_state_scr_ring0_status,
    inp_state_scr_ring1_status,
    inp_state_scr_ring2_status,
    inp_state_scr_ring3_status,
    inp_state_scr_ring4_status,
    inp_state_scr_ring5_status,
}
```

```

inp_state_scr_ring6_status,
inp_state_scr_ring7_status,
inp_state_scr_ring8_status,
inp_state_scr_ring9_status,
inp_state_scr_ring10_status,
inp_state_scr_ring11_status,
inp_state_fci_not_empty,
inp_state_fci_full
}

```

4.4.2 Synchronization Functions

This section describes the synchronization functions. [Table 18](#) lists these functions.

Table 18. Synchronization Functions Summary

Name (args)	Description
SIGNAL_MASK __signals();	Takes a variable length argument list and returns a signal masks for use with ctx_arb.
int signal_test(SIGNAL *sig);	Tests whether or not a signal is set. If the signal is set, it is cleared as a side effect of testing it and a value of 1 is returned.
ctx_swap();	This is a macro that calls the function ctx_wait(voluntary), which voluntarily swaps out the current context.
void ctx_wait(signal_t sig);	Swaps out the current context and waits for the specified signal, either kill, bpt, voluntary, or no_load.
void __wait_for_any();	Generates a ctx_arb on the set of signals specified by the arguments.
void __wait_for_all();	Generates a ctx_arb on the set of signals specified by the arguments.
void signal_same_ME(unsigned int sig_no, unsigned int ctx);	Raise sig_no in ctx of the same ME.
void signal_same_ME_next_ctx(unsigned int sig_no);	Raise sig_no in next context number of the same ME.
void signal_prev_ME(unsigned int sig_no, unsigned int ctx);	Raise sig_no in ctx of the previous ME.
void signal_prev_ME_this_ctx(unsigned int sig_no);	Raise sig_no in the same context number of the previous ME.
void signal_next_ME(unsigned int sig_no, unsigned int ctx);	Raise sig_no in ctx of the next ME.
void signal_next_ME_this_ctx(unsigned int sig_no);	Raise sig_no in the same context number of the next ME.

4.4.2.1 `__signals()`

Function Syntax:

```
SIGNAL_MASK __signals();
```

Description:

This function takes a variable length argument list and is useful for generating signal masks for use with `ctx_arb`. Each argument must be one of the following:

- An address of a `SIGNAL` or `SIGNAL_PAIR` variable. For signal pairs, both even and odd signals are included in the mask
- A signal mask

The function returns a 16 bit signal mask as an `int` that can be used to generate a `ctx_arb` either through the `__wait_for_all()/__wait_for_any()` intrinsics, or through inline assembly.

This function returns a mask representing the hardware signals allocated to your signal variables. Any number of signals of type `SIGNAL` between 1 and 15 can be specified.

Note: When a user passes in a signal mask to a function as an `int` parameter, the compiler cannot always figure out what bits are set in the mask and hence does not know the members of the signal set that is represented by the mask. The compiler does not know the life range of these signals represented in such a mask. To get register allocation right, you have to insert calls to `implicit_read()` (see example below).

```
SIGNAL sig1, sig2;
int mask;
mask = 0
foo( sig1, mask);
mask = __ signals(&sig1, &sig2);
foo(sig2, mask);
__implicit_read(&sig1); //User needs to insert this since
                        //ixp_buf_read() reads the
                        //signals in the mask
__implicit_read(&sig2); //User needs to insert this
                        //since ixp_buf_read() reads the
                        //signals in the mask
```

Arguments:

<variable list> Up to 15 signals.

4.4.2.2 `signal_test()`

Function Syntax:

```
int signal_test(  
    SIGNAL *sig);
```

Description:

This function tests whether or not a signal is set. Signal is a user signal variable. If the signal is set, it gets cleared as a side effect of testing it and a value of 1 is returned by this function. Otherwise, a 0 is returned.

Arguments:

sig1	User signal variable.
------	-----------------------

4.4.2.3 ctx_swap()

Function Syntax:

```
void ctx_swap();
```

Description:

This is a C macro that serves as a wrapper around the function call `ctx_wait(voluntary)`. The `ctx_wait(voluntary)` function call voluntarily swaps out the current context. The context is ready to run again immediately.

Arguments:

None.

4.4.2.4 `ctx_wait()`

Function Syntax:

```
void ctx_wait(signal_t sig);
```

Description:

This function swaps out the current context and waits for the specified signal, bpt, voluntary, kill, or no_load (where the signal mask is specified earlier in one of the local csr CTX_WAKEUP_EVENTS). Please use the `__wait_for_all()`, `__wait_for_any()`, or `signal_test()` intrinsics to wait for signal variables. The `sig` argument is one of kill, voluntary, bpt or no_load.

Arguments:

<code>sig</code>	Special signal to wait for.
------------------	-----------------------------

4.4.2.5 `__wait_for_any()`, `__wait_for_all()`

Function Syntax:

```
void __wait_for_any();  
void __wait_for_all();
```

Description:

These two functions take a variable length argument list. Each argument must be one of:

- Address of a SIGNAL or SIGNAL_PAIR variable. For signal pairs, both even and odd signals are included in the mask used for the `ctx_arb`.
- Return value from a call to `__signals()` or an integer variable that was assigned such a return value.

These functions generate a `ctx_arb` on the set of signals specified by the arguments. In cases where one or more arguments is not a direct SIGNAL/SIGNAL_PAIR address, the indirect form of the `ctx_arb` may be generated. The AND form of the `ctx_arb` is generated from a `__wait_for_all()`, whereas `__wait_for_any()` generates the OR form.

Arguments:

<variable length args> SIGNAL or SIGNAL_PAIR variable, or SIGNAL_MASK returned from the `__signals()` intrinsic.

4.4.2.6 `signal_same_ME`

Function Syntax:

```
void signal_same_ME(  
    unsigned int sig_no,  
    unsigned int ctx);
```

Description:

Raise the specified signal number (`sig_no`) in the specified context (`ctx`) of the same Microengine.

Note: A call to the `__implicit_write()` intrinsic is required immediately after a call to this intrinsic. The Microengine C compiler can't automatically add a call to the `__implicit_write` intrinsic because the `sig_no` may not always be constant and only you know precisely which signal is implicated.

Arguments:

<code>sig_no</code>	The signal number to raise.
<code>ctx</code>	The context in which the signal is to be raised. This context must be running on the same Microengine.

4.4.2.7 `signal_same_ME_next_ctx`

Function Syntax:

```
void signal_same_ME_next_ctx(  
    unsigned int sig_no,);
```

Description:

Raise the specified signal number (`sig_no`) in the next context number of the same Microengine.

Note: A call to the `__implicit_write()` intrinsic is required immediately after a call to this intrinsic. The Microengine C compiler can't automatically add a call to the `__implicit_write` intrinsic because the `sig_no` may not always be constant and only you know precisely which signal is implicated.

Example:

```
sig_no = __signal_number(&sig);  
signal_same_ME(sig_no, 4);  
__implicit_write(&sig); //
```

Arguments:

`sig_no` The signal number to raise.

4.4.2.8 signal_prev_ME

Function Syntax:

```
void signal_prev_ME(  
    unsigned int sig_no,  
    unsigned int ctx);
```

Description:

Raise the specified signal number (sig_no) in the specified context (ctx) of the previous Microengine.

Note: A call to the `__implicit_write()` intrinsic is required immediately after a call to this intrinsic. The Microengine C compiler can't automatically add a call to the `__implicit_write` intrinsic because the `sig_no` may not always be constant and only you know precisely which signal is implicated.

Arguments:

sig_no	The signal number to raise.
ctx	The context in which to raise the signal.

4.4.2.9 `signal_prev_ME_this_ctx`

Function Syntax:

```
void signal_prev_ME_this_ctx(  
    unsigned int sig_no);
```

Description:

Raise the specified signal number (`sig_no`) in the same context number of the previous Microengine.

Note: A call to the `__implicit_write()` intrinsic is required immediately after a call to this intrinsic. The Microengine C compiler can't automatically add a call to the `__implicit_write` intrinsic because the `sig_no` may not always be constant and only you know precisely which signal is implicated.

Arguments:

<code>sig_no</code>	The signal number to raise.
---------------------	-----------------------------

4.4.2.10 `signal_next_ME`

Function Syntax:

```
void signal_next_ME(  
    unsigned int sig_no,  
    unsigned int ctx);
```

Description:

Raise the specified signal number (`sig_no`) in the specified context (`ctx`) of the next Microengine.

Note: A call to the `__implicit_write()` intrinsic is required immediately after a call to this intrinsic. The Microengine C compiler can't automatically add a call to the `__implicit_write` intrinsic because the `sig_no` may not always be constant and only you know precisely which signal is implicated.

Arguments:

<code>sig_no</code>	The signal number to raise.
<code>ctx</code>	The context in which to raise the signal.

4.4.2.11 `signal_next_ME_this_ctx`

Function Syntax:

```
void signal_next_ME_this_ctx(  
    unsigned int sig_no);
```

Description:

Raise the specified signal number (`sig_no`) in the same context number of the next Microengine.

Note: A call to the `__implicit_write()` intrinsic is required immediately after a call to this intrinsic. The Microengine C compiler can't automatically add a call to the `__implicit_write` intrinsic because the `sig_no` may not always be constant and only you know precisely which signal is implicated.

Arguments:

<code>sig_no</code>	The signal number to raise.
---------------------	-----------------------------

4.5 Control and Status Register (CSR) Access Functions

This section discusses the data types and functions used to access the CSR registers.

4.5.1 CAP Data Types

4.5.1.1 cap_csr_t

This enumeration type defines the CAP Control and Status Registers (CSRs) used with the `cap_read()`, `cap_write()`, and `cap_fast_write()` functions. CAP CSRs are chip wide and provide special interprocessor communication features to allow flexible and efficient inter-Microengine and Microengine to core communication. The enumeration values are shown below

Note: In the following enumeration, the # symbol represents a Microengine cluster number and is either 0 or 1; \$ is a Microengine number between 0 and 7; and & is a context number between 0 and 7. For `csr_thd_msg_summary_#_n`, the n represents a register number from 0 to 1. For `csr_scratch_ring_base_n`, n refers to a ring number between 0 and 15.

```
enum {
    csr_thd_msg
    csr_thd_summary_#_$_&<n>,           < n = 0, ..., 1 >
    csr_thd_msg_clr_#_$_&
    csr_thd_msg_#_$_&
    csr_self_destruct_0,
    csr_self_destruct_1,
    csr_interthread_sig,
    csr_thread_interrupt_a
    csr_thread_interrupt_b
    csr_scratch_ring_base<n>,           < n = 0, ..., 15 >
    csr_scratch_ring_head_<n>,         < n = 0, ..., 15 >
    csr_scratch_ring_<n>_tail,         < n = 0, ..., 15 >
    csr_hash_multiplier_48_0,
    csr_hash_multiplier_48_1,
    csr_hash_multiplier_64_0,
    csr_hash_multiplier_64_1,
    csr_hash_multiplier_128_0,
    csr_hash_multiplier_128_1,
    csr_hash_multiplier_128_2,
    csr_hash_multiplier_128_3,
    csr_product_id,
    csr_misc_control,
    csr_perf_counter_control,
    csr_ixp_reset_0
    csr_ixp_reset_1
    csr_clock_control
    csr_strap_options
} cap_csr_t;
```

4.5.1.2 local_csr_t

This enumeration specifies the CSRs used in `local_csr_read` and `local_csr_write` functions. The enumeration values are:

```
enum {
    local_csr_ustore_address,
    local_csr_ustore_data_lower,
    local_csr_ustore_data_upper,
    local_csr_ustore_error_status,
    local_csr_alu_out,
    local_csr_ctx_arb_cntl,
    local_csr_ctx_enables,
    local_csr_cc_enable,
    local_csr_csr_ctx_pointer,
    local_csr_indirect_ctx_sts,
    local_csr_active_ctx_sts,
    local_csr_indirect_ctx_sig_events,
    local_csr_active_ctx_sig_events,
    local_csr_indirect_ctx_wakeup_events,
    local_csr_active_ctx_wakeup_events,
    local_csr_indirect_ctx_future_count,
    local_csr_active_ctx_future_count,
    local_csr_indirect_lm_addr_0,
    local_csr_indirect_lm_addr_0_byte_index,
    local_csr_active_lm_addr_0,
    local_csr_active_lm_addr_0_byte_index,
    local_csr_indirect_lm_addr_1,
    local_csr_indirect_lm_addr_1_byte_index,
    local_csr_active_lm_addr_1,
    local_csr_active_lm_addr_1_byte_index,
    local_csr_byte_index,
    local_csr_t_index,
    local_csr_t_index_byte_index,
    local_csr_indirect_future_count_signal,
    local_csr_nn_get,
    local_csr_nn_put,
    local_csr_timestamp_high,
    local_csr_timestamp_low,
    local_csr_next_neighbor_signal,
    local_csr_prev_neighbor_signal,
    local_csr_same_me_signal,
    local_csr_crc_remainder,
    local_csr_profile_count,
    local_csr_pseudo_random_number,
    local_csr_status,
    local_csr_crc_remainder,
    local_csr_profile_count,
    local_csr_pseudo_random_number,
    local_csr_local_csr_status,
    local_csr_datapath_signature,
    local_csr_datapath_signature_enable
} local_csr_t;
```

4.5.1.3 cap_read_write_ind_t

This structure provides additional or overriding qualifiers on CAP read and write operations with the indirect_ref attribute. For further details on this indirect qualifier, refer to the *LXP2400/LXP2800 Network Processor Programmer's Reference Manual*. This structure has the following bitfields, all of which are unsigned ints:

Field Name	Size	Description
ov_ueng_addr	1	1 to use the microengine specified in the ueng_addr field; 0 to use the issuing microengine (this should never be set)
ueng_addr	5	Specifies the microengine where the result is to be written and signaled upon completion.
ov_ref_count	1	1 to use the count specified in the ref_count field; 0 to use the count argument to the function.
ref_count	4	Reference count indicating the number of longwords to read or write. The value encoded in this field is one less than the reference count. Hence, valid values are 0 - 15.
reserved	9	Unused.
xadd	7	The starting transfer register.
ov_xadd	1	1 to override the transfer register (this should never be set).
ov_ctx	1	1 to use the context specified in the ctx field; 0 to use the current context.
ctx	3	Specifies context where result will be written and signaled upon completion.

4.5.2 CAP Functions

This section describes the CSR access functions that are used to move data between CAP CSRs and a microengine. Table 19 provides a summary.

Table 19. CSR Access Functions Summary (Sheet 1 of 2)

Name (args)	Description
void cap_csr_read[_D](__declspec([sram, dram]_read_reg) void *data, cap_csr_t csr, sync_t sync, SIGNAL *sig_ptr);	Reads the value of the specified CSR into the transfer register specified by data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers.
void cap_csr_read[_D]_ind(__declspec([sram, dram]_read_reg) void *data, cap_csr_t csr, cap_csr_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);	Reads the value of the specified CSR into the transfer register specified by data. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The ind argument provides additional parameters and overrides.
void cap_csr_write[_D](__declspec([sram, dram]_write_reg) void *data, cap_csr_t csr, sync_t sync, SIGNAL *sig_ptr);	Writes the content of data to the specified CSR. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.
void cap_csr_write[_D]_ind(__declspec([sram, dram]_write_reg) void *data, cap_csr_t csr, cap_csr_read_write_ind_t ind, sync_t sync, SIGNAL *sig_ptr);	Writes the content of data to the specified CSR. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides additional parameters and overrides.
void cap_read[_D](__declspec([sram, dram]_read_reg) void *data, unsigned int address, unsigned int count, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);	Reads from the specified CAP address into the data buffer. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers.
void cap_read[_D]_ind(__declspec([sram, dram]_read_reg) void *data, unsigned int address, unsigned int max_nn, cap_read_write_ind_t ind, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);	Reads from the specified CAP address into the data buffer. The ind argument provides additional parameters and overrides. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers.
void cap_write[_D](__declspec([sram, dram]_write_reg) void *data, unsigned int address, unsigned int count, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);	Writes from the data buffer to the specified CAP address. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.
void cap_write[_D]_ind(__declspec([sram, dram]_write_reg) void *data, unsigned int address, unsigned int max_nn, cap_read_write_ind_t ind, reflect_sig_t reflect_sig, sync_t sync, SIGNAL *sig_ptr);	Writes from the data buffer to the specified CAP address. The _D version of this intrinsic must be used if the data argument is in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides additional parameters and overrides.

Table 19. CSR Access Functions Summary (Continued) (Sheet 2 of 2)

Name (args)	Description
void cap_fast_write(unsigned int data, cap_csr_t csr);	Writes the content of data to the specified CSR and eliminates the need to pull data from a transfer register during a write operation and therefore reduces the time required to complete the write operation.
unsigned int local_csr_read(local_csr_t csr);	Reads and returns the value of the specified local CSR.
void local_csr_write(local_csr_t csr, unsigned int data);	Writes the content of data to the specified local CSR.

4.5.2.1 cap_csr_read(), cap_csr_read_D()

Function Syntax:

```
void cap_csr_read[_D] (  
    __declspec([sram, dram]_read_reg) void *data,  
    cap_csr_t csr,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

These functions read from the specified CSR and write the data into the transfer register specified by data. The cap_csr_read() function writes the data into an SRAM transfer register while the cap_csr_read_D() function writes the data into a DRAM transfer register. Arguments csr and sync should be constant literals as required by the microcode assembler. The csr argument should refer to an appropriate CSR that can be specified on a CAP read microcode instruction. The sig_ptr argument identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data buffer to read into.
csr	CSR to read. This argument must be a constant.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.5.2.2 cap_csr_read_ind(), cap_csr_read_D_ind()

Function Syntax:

```
void cap_csr_read[_D]_ind(
    __declspec( [sram, dram]_read_reg) void *data,
    cap_csr_t csr,
    cap_csr_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read from the specified CSR and write the data into the transfer register specified by data. The cap_csr_read_ind() function writes the data into an SRAM transfer register while the cap_csr_read_D_ind() function writes the data into a DRAM transfer register. Arguments csr and sync should be constant literals as required by the microcode assembler. The csr argument should refer to an appropriate CSR that can be specified on a CAP read microcode instruction. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the cap_csr_read_write_ind_t data type. The sig_ptr argument identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

Arguments:

data	Data buffer to read into.
csr	CSR to read. This argument must be a constant.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.5.2.3 cap_csr_write(), cap_csr_write_D()

Function Syntax:

```
void cap_csr_write[_D] (  
    __declspec([sram, dram]_write_reg) void *data,  
    cap_csr_t csr,  
    sync_t sync,  
    SIGNAL *sig_ptr);
```

Description:

This function writes the contents of data to the specified CSR. Arguments csr and sync should be constant literals as required by the microcode assembler. The csr argument should refer to an appropriate CSR that can be specified on a CAP write microcode instruction. The sig_ptr argument identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data	Data buffer to read from.
csr	CSR to write to.
sync	Type of synchronization to use. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.5.2.4 cap_csr_write_ind(), cap_csr_write_D_ind()

Function Syntax:

```
void cap_csr_write[_D]_ind(
    __declspec([sram, dram]_write_reg) void *data,
    cap_csr_t csr,
    cap_csr_read_write_ind_t ind,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes the contents of the buffer data to the specified CSR. Arguments `csr` and `sync` should be constant literals as required by the microcode assembler. The `csr` argument should refer to an appropriate CSR that can be specified on a CAP write microcode instruction. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `cap_csr_read_write_ind_t` data type. The `sig_ptr` argument identifies your signal variable to use for signaling event completion.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

<code>data</code>	Data buffer to read from.
<code>csr</code>	CSR to write to.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Signal to raise upon completion.

4.5.2.5 cap_read(), cap_read_D()

Function Syntax:

```
void cap_read[_D] (
    __declspec([sram, dram]_read_reg) void *data,
    unsigned int address,
    unsigned int count,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read data from the specified address on a CAP subunit into a transfer register specified by the data argument. The cap_read() function reads the data into an SRAM transfer register while the cap_read_D() function reads the data into a DRAM transfer register. The address argument should refer to an appropriate CAP subunit that can be specified in a CAP read microcode instruction. The count argument must be in the range of 1 through 16. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref resulting in a loss of performance. Arguments reflect_sig and sync should be constant literals as required by the microcode assembler.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

If these functions are used to access CSRs (Control Status Registers) in the CAP subunit, the “count” parameter must be 1. Only one CSR can be accessed at a time.

Arguments:

data	Data buffer to read into.
address	Address to read from.
count	Number of longwords to read in the range of 1 - 16.
reflect_sig	Reflect signal delivery type.
sync	Type of synchronization to use. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.5.2.6 cap_read_ind(), cap_read_D_ind()

Function Syntax:

```
void cap_read[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data,
    unsigned int address,
    unsigned int max_nn,
    cap_read_write_ind_t ind,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

These functions read data from the specified address on a CAP subunit into a transfer register specified by the data argument. The cap_read_ind() function reads the data into an SRAM transfer register while the cap_read_D_ind() function reads the data into a DRAM transfer register. Arguments reflect_sig and sync should be constant literals as required by the microcode assembler. The max_nn argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the ind argument specifies the exact number of longwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of longwords to be transferred and must be given as 8 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the cap_read_write_ind_t data type. The sig_ptr argument identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available in 8-context mode.

If these functions are used to access CSRs (Control Status Registers) in the CAP subunit, the “max_nn” parameter must be 1, and the “count” field of the indirect access structure must also be 1. Only one CSR can be accessed at a time.

Arguments:

data	Data buffer to read into.
address	Address to read from.
max_nn	Maximum number of longwords to read in the range of 1 - 16.
ind	Indirect word.
reflect_sig	Reflect signal deliver type.
sync	Type of synchronization to use. This argument must be a constant. Note that if 'sig_done' synchronization is used, a call to the __implicit_read() intrinsic may be required if not all the data is used. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.5.2.7 cap_write(), cap_write_D()

Function Syntax:

```
void cap_write[_D](
    __declspec([sram, dram]_write_reg) void *data,
    unsigned int address,
    unsigned int count
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes the contents of data to the specified CAP address. Argument count is preferred to be a constant; otherwise, the compiler generates an indirect_ref resulting in a loss of performance. The count argument must be in the range of 1 through 16. Arguments reflect_sig and sync should be constant literals as required by the microcode assembler. The address argument should refer to an appropriate CAP subunit that can be specified on a CAP write microcode instruction. The sig_ptr argument identifies your signal variable to use for signaling event completion.

Note: The _D version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

If these functions are used to access CSRs (Control Status Registers) in the CAP subunit, the “count” parameter must be 1. Only one CSR can be accessed at a time.

Arguments:

data	Data buffer to write to.
address	Address to read from.
count	Number of longwords to write in the range of 1 - 16.
reflect_sig	Reflect signal delivery type.
sync	Type of synchronization to use. If 'sig_done' synchronization is used, a call to either the __implicit_read() or __free_write_buffer() intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Signal to raise upon completion.

4.5.2.8 cap_write_ind(), cap_write_D_ind()

Function Syntax:

```
void cap_write[_D]_ind(
    __declspec([sram, dram]_write_reg) void *data,
    unsigned int address,
    unsigned int max_nn,
    cap_read_write_ind_t ind,
    reflect_sig_t reflect_sig,
    sync_t sync,
    SIGNAL *sig_ptr);
```

Description:

This function writes the contents of data to the specified CAP address. Arguments `reflect_sig` and `sync` should be constant literals as required by the microcode assembler. The address argument should refer to an appropriate CAP subunit that can be specified on a CAP write microcode instruction. The `max_nn` argument must be a constant in the range of 1 through 16 and specifies the maximum number of longwords to be transferred while the `ind` argument specifies the exact number of longwords to be transferred. If the `ind` argument does not specify a count, then `max_nn` represents the number of longwords to be transferred and must be given as 8 or less. The `ind` argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the `cap_read_write_ind_t` data type. The `sig_ptr` argument identifies your signal variable to use for signaling event completion.

Note: The `_D` version of the intrinsic must be used if the data argument is in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

If these functions are used to access CSRs (Control Status Registers) in the CAP subunit, the “`max_nn`” parameter must be 1, and the “count” field of the indirect access structure must also be 1. Only one CSR can be accessed at a time.

Arguments:

<code>data</code>	Data buffer to write to.
<code>address</code>	Address to read from.
<code>max_nn</code>	Maximum number of longwords to write in the range of 1 - 16.
<code>reflect_sig</code>	Reflect signal delivery type.
<code>ind</code>	Indirect word.
<code>sync</code>	Type of synchronization to use. This argument must be a constant. If 'sig_done' synchronization is used, a call to either the <code>__implicit_read()</code> or <code>__free_write_buffer()</code> intrinsics is required to prevent the transfer registers from being reused by the compiler before the operation has completed. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
<code>sig_ptr</code>	Pointer to signal to use.

4.5.2.9 cap_fast_write()

Function Syntax:

```
void cap_fast_write(  
    unsigned int data,  
    cap_csr_t csr);
```

Description:

This function writes the contents of data to the specified CSR, but eliminates the need to pull data from a transfer register during a write operation and therefore reduces the time required to complete the write operation. Arguments csr should be a constant literal as required by the microcode assembler. The csr argument should refer to an appropriate CSR that can be specified on a CAP fast write microcode instruction.

Note: The scratch ring CSRs cannot be written to using this function. This is a hardware restriction.

Arguments:

data	Data buffer to read from.
csr	CSR to write to.

4.5.2.10 local_csr_read()

Function Syntax:

```
unsigned int local_csr_read(  
    local_csr_t csr);
```

Description:

This function reads and returns the value of the specified local CSR. The `csr` argument should be a constant literal that refers to an appropriate local CSR and that can be specified on a LOCAL_CSR_RD microcode instruction.

Arguments:

<code>csr</code>	CSR to read from.
------------------	-------------------

4.5.2.11 local_csr_write()

Function Syntax:

```
void local_csr_write(  
    local_csr_t csr,  
    unsigned int data);
```

Description:

This function writes the contents of data to the specified local CSR. The csr argument should be a constant literal that refers to an appropriate local CSR and that can be specified on a LOCAL_CSR_WR microcode instruction.

Arguments:

csr	CSR to write to.
data	Data to write to the specified CSR.

4.6 Hash Access Functions

The Hash Access functions allow you to perform 48, 64, and 128 bit operations on the hash unit.

4.6.1 Data Types

The `hash_ind_t` structure provides additional or overriding qualifiers on Hash unit operations with the `indirect_ref` attribute. For further details on this indirect qualifier, refer to the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

Note: Since the compiler does all register allocation, including all transfer registers, it is an error to set the `ov_xadd` field to override the transfer register allocated and specified in the instruction by the compiler. For this same reason, it is also an error to set the `ov_ueng_addr` fields.

The `hash_ind_t` structure has the following bitfields:

Field Name	Size	Description
<code>ctx</code>	3	Specifies context where result will be written and signaled upon completion.
<code>ov_ctx</code>	1	1 to use the context above, 0 to use the current context.
<code>ov_xadd</code>	1	1 to override the transfer register (this should never be set)
<code>xadd</code>	7	The starting transfer register.
<code>reserved1</code>	9	Unused.
<code>hash_count</code>	2	Hash count 1, 2, or 3.
<code>reserved2</code>	2	Unused.
<code>ov_hash_count</code>	1	1 to use the count above, 0 to use the count argument to the function.
<code>ueng_addr</code>	5	Specifies ME where result will be written and signaled upon completion.
<code>ov_ueng_addr</code>	1	1 to use <code>ueng_addr</code> above, 0 to use the current <code>ueng_addr</code> (this should never be set).

4.6.2 Functions

This section describes the hash access functions. [Table 20](#) summarizes these functions.

Table 20. Hash Functions Summary

Name (args)	Description
<pre>void hash_48[_D](__declspec([sram, dram]_read_reg) void *data_out, __declspec([sram, dram]_write_reg) void *data_in, unsigned int count, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Computes a 48-bit hash on up to count quadwords of data_in. Count must be a compile time constant between 1 and 3. The _D version of this intrinsic must be used if the data_out and data_in arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void hash_48[_D]_ind(__declspec([sram, dram]_read_reg) void *data_out, __declspec([sram, dram]_write_reg) void *data_in, unsigned int max_nn, hash_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Computes a 48-bit hash on up to max_nn quadwords of data_in. The _D version of this intrinsic must be used if the data_out and data_in arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides overrides and additional parameters.</p>
<pre>void hash_64[_D](__declspec([sram, dram]_read_reg) void *data_out, __declspec([sram, dram]_write_reg) void *data_in, unsigned int count, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Computes a 64-bit hash on up to count quadwords of data_in. Count must be a compile time constant between 1 and 3. The _D version of this intrinsic must be used if the data_out and data_in arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void hash_64[_D]_ind(__declspec([sram, dram]_read_reg) void *data_out, __declspec([sram, dram]_write_reg) void *data_in, unsigned int max_nn, hash_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Computes a 64-bit hash on up to max_nn quadwords of data_in. The _D version of this intrinsic must be used if the data_out and data_in arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides overrides and additional parameters.</p>
<pre>void hash_128[_D](__declspec([sram, dram]_read_reg) void *data_out, __declspec([sram, dram]_write_reg) void *data_in, unsigned int count, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Computes a 128-bit hash on up to 2*count quadwords of data_in. Count must be a compile time constant between 1 and 3. The _D version of this intrinsic must be used if the data_out and data_in arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode.</p>
<pre>void hash_128[_D]_ind(__declspec([sram, dram]_read_reg) void *data_out, __declspec([sram, dram]_write_reg) void *data_in, unsigned int max_nn, hash_ind_t ind, sync_t sync, SIGNAL_PAIR *sig_ptr);</pre>	<p>Computes a 128-bit hash on up to 2*max_nn quadwords of data_in. The _D version of this intrinsic must be used if the data_out and data_in arguments are in DRAM transfers registers. The _D version is only available on IXP28XX hardware in 8-context mode. The ind argument provides overrides and additional parameters.</p>

4.6.2.1 hash_48(), hash_48_D()

Function Syntax:

```
void hash_48[_D] (
    __declspec([sram, dram]_read_reg) void *data_out,
    __declspec([sram, dram]_write_reg) void *data_in,
    unsigned int count,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function computes a 48-bit hash on count quadwords of data_in. Count must be a compile time constant between 1 and 3. The sync argument must be a constant. The results are placed in data_out. The signal used for synchronization is addressed by sig_ptr.

Note: The _D version of the intrinsic must be used when both the data_out and data_in arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data_out	Hashed data.
data_in	Data to hash.
count	Number of quadwords of data.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Address of user signal variable.

4.6.2.2 hash_48_ind(), hash_48_D_ind()

Function Syntax:

```
void hash_48[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data_out,
    __declspec([sram, dram]_write_reg) void *data_in,
    unsigned int max_nn,
    hash_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function computes a 48-bit hash on up to max_nn quadwords of data_in. The results are placed in data_out. The max_nn argument must be a constant in the range of 1 through 3 and specifies the maximum number of quadwords to be transferred while the ind argument specifies the exact number of quadwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of quadwords to be transferred and must be given as 3 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the hash_ind_t data type. The signal used for synchronization is addressed by sig_ptr.

Note: The _D version of the intrinsic must be used when both the data_out and data_in arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data_out	Hashed data.
data_in	Data to hash.
max_nn	Number of quadwords of data in the range of 1 - 3.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Address of user signal variable.

4.6.2.3 hash_64(), hash_64_D()

Function Syntax:

```
void hash_64[_D](
    __declspec([sram, dram]_read_reg) void *data_out,
    __declspec([sram, dram]_write_reg) void *data_in,
    unsigned int count,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function computes a 64-bit hash on up to count quadwords of data_in. Argument count must be a compile time constant between 1 and 3. Argument sync must be constant. The results are placed in data_out. The signal used for synchronization is addressed by sig_ptr.

Note: The _D version of the intrinsic must be used when both the data_out and data_in arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data_out	Hashed data.
data_in	Data to hash.
count	Number of quadwords of data.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Address of user signal variable.

4.6.2.4 hash_64_ind(), hash_64_D_ind()

Function Syntax:

```
void hash_64[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data_out,
    __declspec([sram, dram]_write_reg) void *data_in,
    unsigned int max_nn,
    hash_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function computes a 64-bit hash on up to max_nn quadwords of data_in. The results are placed in data_out. The max_nn argument must be a constant in the range of 1 through 3 and specifies the maximum number of quadwords to be transferred while the ind argument specifies the exact number of quadwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of quadwords to be transferred and must be given as 3 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the hash_ind_t data type. The signal used for synchronization is addressed by sig_ptr.

Note: The _D version of the intrinsic must be used when both the data_out and data_in arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data_out	Hashed data.
data_in	Data to hash.
max_nn	Number of quadwords of data in the range of 1 - 3.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Address of user signal variable.

4.6.2.5 hash_128(), hash_128_D()

Function Syntax:

```
void hash_128[_D](
    __declspec([sram, dram]_read_reg) void *data_out,
    __declspec([sram, dram]_write_reg) void *data_in,
    unsigned int count,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function computes a 128-bit hash on 2 times count quadwords of data_in. Count must be a compile time constant between 1 and 3. The sync argument must be a constant. The results are placed in data_out. The signal used for synchronization is addressed by sig_ptr.

Note: The _D version of the intrinsic must be used when both the data_out and data_in arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data_out	Hashed data.
data_in	Data to hash.
count	Number of 2 times quadwords of data.
sync	Type of synchronization to use. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Address of user signal variable.

4.6.2.6 hash_128_ind(), hash_128_D_ind()

Function Syntax:

```
void hash_128[_D]_ind(
    __declspec([sram, dram]_read_reg) void *data_out,
    __declspec([sram, dram]_write_reg) void *data_in,
    unsigned int max_nn,
    hash_ind_t ind,
    sync_t sync,
    SIGNAL_PAIR *sig_ptr);
```

Description:

This function computes a 128-bit hash on up to 2 times max_nn quadwords of data_in. The results are placed in data_out. The max_nn argument must be a constant in the range of 1 through 3 and specifies the maximum number of quadwords to be transferred while the ind argument specifies the exact number of quadwords to be transferred. If the ind argument does not specify a count, then max_nn represents the number of quadwords to be transferred and must be given as 3 or less. The ind argument provides additional parameters and overrides. There are restrictions on the value specified in the override as noted in the description of the hash_ind_t data type. The signal used for synchronization is addressed by sig_ptr.

Note: The _D version of the intrinsic must be used when both the data_out and data_in arguments are in DRAM transfer registers. This version is only available on IXP28XX Rev. B hardware in 8-context mode.

Arguments:

data_out	Hashed data.
data_in	Data to hash.
max_nn	Number of 2 times quadwords of data in the range of 1 - 3.
ind	Indirect word.
sync	Type of synchronization to use. This argument must be a constant. The synchronization argument must be 'sig_done'. A call to either the __implicit_read() or __free_write_buffer() intrinsics is required on the write transfer register argument to prevent the registers from being reused before the operation has completed. A call to __implicit_read() on the read transfer register argument will be needed if not all the data is read. Please refer to Section 7.2, “Things to Remember When Writing Microengine C Code” for details and an example.
sig_ptr	Address of user signal variable.

4.6.3 Limitations on Hash Functions

Since the hardware HASH instruction takes only one operand for both source and destination operand, the compiler allocates the read transfer register pointed to by `data_in` and the write transfer register pointed to by `data_out` to the same physical register number. As a result, certain usage is not valid. For example:

```
__declspec(sram_read_reg) long long r1[2], r2, r3;
__declspec(sram_write_reg) long long w1, w2, w3;
SIGNAL s1;
...
hash_48(&r1[0], &w1, 1, sig_done, &s1); // r1[0] and w1 are mapped to the same
// physical register number

__wait_for_all(&s1);

hash_48(&r1[1], &w1, 1, sig_done, &s1); // Error: r1[1] and w1 can not be
//mapped to the same physical
//register number

__wait_for_all(&s1);
```

Another case:

```
hash_48(&r2, &w1, &w1, 1, sig_done, &s1); // r2 and w1 are mapped to the same
// physical register number

__wait_for_all(&s1);
hash_48( &r3, &w1, &w2, 1, sig_done, &s1); // r3 and w1 are mapped to the same
// physical register number

__wait_for_all(&s1);
__asm alu[X, --, B, r2] // Error: the second hash_48 corrupted
// r2 because r2 and r3 are mapped to the same
// physical register.
```

4.7 CAM (Content Addressable Memory) Access Functions

The data types and functions described in this section are for accessing the CAM features of the Network Processor. CAM is a hardware feature where a content match is performed to get an index to associated information.

4.7.1 Data Types

4.7.1.1 `cam_lookup_t`

The `cam_lookup_t` structure is used to capture the results of a CAM lookup. It has the following bit-fields:

Field Name	Size	Description
<code>zeroes1</code>	3	Reserved bits that are set to 0.
<code>entry_num</code>	4	Matched CAM entry number on hit. LRU entry number on miss.
<code>hit</code>	1	1 indicates a hit; 0 indicates a miss
<code>state</code>	4	State bits associated with matched entry on hit; 0 on miss.
<code>zeroes2</code>	20	Reserved bits that are set to 0.

4.7.2 Functions

This section describes the CAM access functions. [Table 21](#) summarizes this functions.

Table 21. CAM Access Functions Summary

Name (args)	Description
void cam_clear();	Clears all entries in the CAM.
cam_lookup_t cam_lookup(unsigned int lookup_val);	Performs a CAM lookup and returns the hit/miss status, state, and entry number as bitfields in the return value.
unsigned int cam_read_tag(unsigned int entry_num);	Reads out the tag associated with the CAM entry specified by entry_num.
cam_lookup_t cam_read_state(unsigned int entry_num);	Reads out the state associated with the CAM entry specified by entry_num and returns it in the state bit field of the return value of cam_lookup_t.
void cam_write_state(unsigned int entry_num, unsigned int state);	Sets the state for entry_num in the CAM to the value specified in the argument state.
void cam_write(unsigned int index, unsigned int tag, unsigned int state);	Writes an entry in the CAM specified by the argument index with the value specified by tag, and sets the state to the value specified in the state argument.



4.7.2.1 cam_clear()

Function Syntax:

```
void cam_clear();
```

Description:

This function clears all entries in the CAM.

Arguments:

None.

4.7.2.2 `cam_lookup()`

Function Syntax:

```
cam_lookup_t cam_lookup(  
    unsigned int lookup_val);
```

Description:

This function performs a CAM lookup and returns the hit/miss status, state, and entry number as bitfields in the return value. In the event of a miss, the entry value is the LRU (least recently used) entry (which is the suggested entry to replace) and state bits are 0. On a CAM hit, this function has the side effect of marking the CAM entry as MRU (most recently used).

Arguments:

<code>lookup_val</code>	The value to lookup in the CAM.
-------------------------	---------------------------------



4.7.2.3 `cam_read_tag()`

Function Syntax:

```
unsigned int cam_read_tag(  
    unsigned int entry_num);
```

Description:

This function reads out the tag associated with the CAM entry specified by the `entry_num` argument.

Arguments:

<code>entry_num</code>	The CAM entry whose tag is returned.
------------------------	--------------------------------------

4.7.2.4 `cam_read_state()`

Function Syntax:

```
cam_lookup_t cam_read_state(  
    unsigned int entry_num);
```

Description:

This function reads out the state associated with the CAM entry specified by the `entry_num` argument and returns it in the state bitfield of the return value. All other fields of the return value structure are set to 0.

Arguments:

<code>entry_num</code>	The CAM entry whose state is returned.
------------------------	--

4.7.2.5 `cam_write_state()`

Function Syntax:

```
void cam_write_state(  
    unsigned int entry_num,  
    unsigned int state);
```

Description:

This function sets the state for the entry in the CAM specified by the `entry_num` argument to the value specified in the argument `state`. Argument `state` must be a constant literal specified directly in the intrinsic's argument list. Otherwise, the compiler may have to generate runtime checks for the possible 16 values, since the microcode only accepts a constant literal for the state.

Arguments:

<code>entry_num</code>	The CAM entry whose state is set.
<code>state</code>	The state to set for the CAM entry.

4.7.2.6 `cam_write()`

Function Syntax:

```
void cam_write(  
    unsigned int index,  
    unsigned int tag,  
    unsigned int state);
```

Description:

This function writes an entry in the CAM specified by the argument `index` with the value specified by `tag`, and sets the state to the value specified in the argument `state`. Argument `state` must be a constant literal specified directly in the intrinsic's argument list. Otherwise, the compiler may have to generate runtime checks for the possible 16 values, since the microcode only accepts a constant literal for the state.

Arguments:

<code>index</code>	The CAM entry to write
<code>tag</code>	Value to set for this CAM entry.
<code>state</code>	State to set for this CAM entry.

4.8 CRC Access Functions

This section describes the data types and functions that provide access to the Cyclic Redundancy Check unit of the IXP2400/IXP2800 Network Processor.

4.8.1 Data Types

4.8.1.1 bytes_specifier_t

The `bytes_specifier_t` enumeration is used as an argument to the CRC functions and specifies one or more contiguous bytes within a longword of big-endian or little endian data. For example, the `bytes_0_3` item in this enumeration refers to bytes 0 through 3. When using the big endian CRC functions, byte 0 refers to the most significant byte and byte 3 refers to the least significant byte. When using the little endian CRC functions, byte 0 refers to the least significant byte and byte 3 refers to the most significant byte.

Field Name	Description	
	Big Endian	Little Endian
<code>bytes_0_3</code>	0, 1, 2, 3	3, 2, 1, 0
<code>bytes_0_2</code>	0, 1, 2	2, 1, 0
<code>bytes_0_1</code>	0, 1	1, 0
<code>byte_0</code>	0	0
<code>bytes_1_3</code>	1, 2, 3	3, 2, 1
<code>bytes_2_3</code>	2, 3	3, 2
<code>byte_3</code>	3	3

4.8.2 Functions

This sections describes the CRC (Cyclic Redundancy Check) functions. [Table 22](#) summarizes these functions.

Table 22. CRC Access Functions Summary (Sheet 1 of 3)

Name (args)	Description
<pre>unsigned int crc_5_be(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bytes of the data argument that are assumed to be in big endian layout.
<pre>unsigned int crc_5_be_bit_swap(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bits of the data arguments that are assumed to be in big endian layout. The bits in each specified byte of the data argument are swapped before the computation begins.
<pre>unsigned int crc_5_le(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bytes of the data argument, are assumed to be in little endian layout.
<pre>unsigned int crc_5_le_bit_swap(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bits of the data arguments that are assumed to be in little endian layout. The bits in each specified byte of the data argument are swapped before the computation begins.
<pre>unsigned int crc_10_be(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bytes of the data argument that are assumed to be in big endian layout, and returns the result of the computation.
<pre>unsigned int crc_10_be_bit_swap(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bits of the data argument that are assumed to be in big endian layout. The bits in each specified byte of the data argument are swapped before the computation begins.
<pre>unsigned int crc_10_le(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bytes of the data argument that are assumed to be in little endian layout, and returns the result of the computation.
<pre>unsigned int crc_10_le_bit_swap(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bits of the data argument that are assumed to be in little endian layout. The bits in each specified byte of the data argument are swapped before the computation begins.
<pre>unsigned int crc_16_be(unsigned int data, bytes_specifier_t bspec);</pre>	Performs a CRC-CCITT computation on specified bytes in the data argument that is assumed to be in big endian format, and returns the results.
<pre>unsigned int crc_16_be_bit_swap(unsigned int data, bytes_specifier_t bspec);</pre>	Performs a CRC-CCITT computation on specified bits in argument data that is assumed to be in big endian format, and returns the results. The bits in each byte are swapped before the computation begins.
<pre>unsigned int crc_16_le(unsigned int data, bytes_specifier_t bspec);</pre>	Performs a CRC-CCITT computation on specified bytes in the data argument that is assumed to be in little endian format, and returns the results.

Table 22. CRC Access Functions Summary (Continued) (Sheet 2 of 3)

Name (args)	Description
unsigned int crc_16_le_bit_swap(unsigned int data, bytes_specifier_t bspec);	Performs a CRC-CCITT computation on specified bits in argument data that is assumed to be in little endian format, and returns the results. The bits in each byte are swapped before the computation begins.
unsigned int crc_ccitt_be(unsigned int data, bytes_specifier_t bspec);	Performs a CRC-CCITT computation on specified bytes in the data argument that is assumed to be in big endian format, and returns the results.
unsigned int crc_ccitt_be_bit_swap(unsigned int data, bytes_specifier_t bspec);	Performs a CRC-CCITT computation on specified bits in argument data that is assumed to be in big endian format, and returns the results. The bits in each byte are swapped before the computation begins.
unsigned int crc_ccitt_le(unsigned int data, bytes_specifier_t bspec);	Performs a CRC-CCITT computation on specified bytes in the data argument that is assumed to be in little endian format, and returns the results.
unsigned int crc_ccitt_le_bit_swap(unsigned int data, bytes_specifier_t bspec);	Performs a CRC-CCITT computation on specified bits in argument data that is assumed to be in little endian format, and returns the results. The bits in each byte are swapped before the computation begins.
unsigned int crc_32_be(unsigned int data, bytes_specifier_t bspec);	Performs a CRC 32 computation on specified bits in the data argument that is assumed to be in big endian format, and returns the results.
unsigned int crc_32_be_bit_swap(unsigned int data, bytes_specifier_t bspec);	Performs a CRC 32 computation on specified bits in the data argument that is assumed to be in big endian format, and returns the results. The bits in each byte are swapped before the computation begins.
unsigned int crc_32_le(unsigned int data, bytes_specifier_t bspec);	Performs a CRC 32 computation on specified bytes in the data argument that is assumed to be in little endian format.
unsigned int crc_32_le_bit_swap(unsigned int data, bytes_specifier_t bspec);	Performs a CRC 32 computation on specified bits in the data argument that is assumed to be in little endian format. The bits in each byte are swapped before the computation begins.
unsigned int crc_iscsi_be(unsigned int data, bytes_specifier_t bspec);	(IXP28xx Rev. B and above only.) Performs a CRC 32 computation on specified bits in the data argument that is assumed to be in big endian format, and returns the results.
unsigned int crc_iscsi_be_bit_swap(unsigned int data, bytes_specifier_t bspec);	(IXP28xx Rev. B and above only.) Performs a CRC 32 computation on specified bits in the data argument that is assumed to be in big endian format, and returns the results. The bits in each byte are swapped before the computation begins.
unsigned int crc_iscsi_le(unsigned int data, bytes_specifier_t bspec);	(IXP28xx Rev. B and above only.) Performs a CRC 32 computation on specified bytes in the data argument that is assumed to be in little endian format.

Table 22. CRC Access Functions Summary (Continued) (Sheet 3 of 3)

Name (args)	Description
<pre>unsigned int crc_iscsi_le_bit_swap(unsigned int data, bytes_specifier_t bspec);</pre>	(IXP28xx Rev. B and above only.) Performs a CRC 32 computation on specified bits in the data argument that is assumed to be in little endian format. The bits in each byte are swapped before the computation begins.
<pre>unsigned int crc_read();</pre>	Returns the CRC remainder accumulated so far.
<pre>void crc_write(unsigned int residue);</pre>	Initializes the CRC remainder with the value of the residue argument.

4.8.2.1 `crc_5_be()`

Function Syntax:

```
unsigned int crc_5_be(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bytes of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer’s Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 5 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.2 `crc_5_be_bit_swap()`

Function Syntax:

```
unsigned int crc_5_be_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bits of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function. The input argument `data` is returned.

Arguments:

<code>data</code>	The data on which to perform the CRC 5 computation.
-------------------	---

4.8.2.3 `crc_5_le()`

Function Syntax:

```
unsigned int crc_5_le(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bytes of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument, and returns the result of the computation.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 5 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.4 `crc_5_le_bit_swap()`

Function Syntax:

```
unsigned int crc_5_le_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-5 computation on the specified bits of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 5 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.5 `crc_10_be()`

Function Syntax:

```
unsigned int crc_10_be(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bytes of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 10 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.6 `crc_10_be_bit_swap()`

Function Syntax:

```
unsigned int crc_10_be_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bits of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 10 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.7 `crc_10_le()`

Function Syntax:

```
unsigned int crc_10_le(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bytes of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 10 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.8 `crc_10_le_bit_swap()`

Function Syntax:

```
unsigned int crc_10_le_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a CRC-10 computation on the specified bits of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 10 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.9 `crc_16_be()`

Function Syntax:

```
unsigned int crc_16_be(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC-CCITT computation on specified bytes of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer’s Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 16 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.10 `crc_16_be_bit_swap()`

Function Syntax:

```
unsigned int crc_16_be_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC-CCITT computation on specified bits of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 16 computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.11 `crc_16_le()`

Function Syntax:

```
unsigned int crc_16_le(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC-CCITT computation on specified bytes of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer’s Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 16 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.12 `crc_16_le_bit_swap()`

Function Syntax:

```
unsigned int crc_16_le_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC-CCITT computation on specified bits of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 16 computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.13 `crc_ccitt_be`

Function Syntax:

```
unsigned int crc_ccitt_be(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function is equivalent to `crc_16_be()`. It performs a 16-bit CCITT CRC computation on the specified bytes of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.14 `crc_ccitt_be_bit_swap`

Function Syntax:

```
unsigned int crc_ccitt_be_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function is equivalent to `crc_16_be_bit_swap()`. It performs a CCITT CRC-16 computation on the specified bits of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.15 `crc_ccitt_le`

Function Syntax:

```
unsigned int crc_ccitt_le(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function is equivalent to `crc_16_be()`. It performs a 16-bit CCITT CRC computation on the specified bytes of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.16 `crc_ccitt_le_bit_swap`

Function Syntax:

```
unsigned int crc_ccitt_le_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function is equivalent to `crc_16_le_bit_swap()`. It performs a CCITT CRC-16 computation on the specified bits of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.17 `crc_32_be()`

Function Syntax:

```
unsigned int crc_32_be(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC 32 computation on specified bytes of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer’s Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.18 `crc_32_be_bit_swap()`

Function Syntax:

```
unsigned int crc_32_be_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC 32 computation on specified bits of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.19 `crc_32_le()`

Function Syntax:

```
unsigned int crc_32_le(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC 32 computation on specified bytes of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.20 `crc_32_le_bit_swap()`

Function Syntax:

```
unsigned int crc_32_le_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC 32 computation on specified bits of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.21 `crc_iscsi_be()`

Function Syntax:

```
unsigned int crc_iscsi_be(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a 32-bit iSCSI CRC computation on the specified bytes of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.22 `crc_iscsi_be_bit_swap()`

Function Syntax:

```
unsigned int crc_iscsi_be_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a 32-bit iSCSI CRC computation on the specified bits of the data argument that is assumed to be in big endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_BE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function. The input argument `data` is returned.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.23 `crc_iscsi_le()`

Function Syntax:

```
unsigned int crc_iscsi_le(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

(IXP28xx Rev. B and above only) This function performs a 32-bit iSCSI CRC computation on the specified bytes of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer’s Reference Manual*. This function returns the unmodified value of the data argument.

The bytes are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bytes in the data argument on which to perform the computation.

4.8.2.24 `crc_iscsi_le_bit_swap()`

Function Syntax:

```
unsigned int crc_iscsi_le_bit_swap(  
    unsigned int data,  
    bytes_specifier_t bspec);
```

Description:

This function performs a CRC 32 computation on specified bits of the data argument that is assumed to be in little endian layout. The CRC is calculated using a remainder that resides in the CRC_Remainder Local CSR and in the data argument. The CRC_Remainder Local CSR is typically initialized once prior to executing CRC instructions using the `crc_write()` intrinsic (see [Section 4.8.2.26, “`crc_write\(\)`”](#)). To get the results of the CRC computation, use the `crc_read()` intrinsic (see [Section 4.8.2.25, “`crc_read\(\)`”](#)). For more detailed information, refer to the CRC_LE instruction in Section 3 of the *IXP2400/IXP2800 Network Processor Programmer's Reference Manual*. This function returns the unmodified value of the data argument.

The bits in each specified byte of the data argument are swapped before the computation begins. Bit 7 is swapped with bit 0, bit 6 with bit 1, bit 5 with bit 2, and bit 4 with bit 3. The bits are specified by the argument `bspec`, which must be a constant enum literal specified directly in the argument. The previous residue must be set up in the CRC remainder prior to calling this function.

Arguments:

<code>data</code>	The data on which to perform the CRC 32 computation.
<code>bspec</code>	The specified bits in the data argument on which to perform the computation.

4.8.2.25 crc_read()

Function Syntax:

```
unsigned int crc_read();
```

Description:

This function returns the CRC remainder accumulated so far.

Arguments:

None.

4.8.2.26 `crc_write()`

Function Syntax:

```
void crc_write(  
    unsigned int residue);
```

Description:

This function initializes the CRC remainder with the argument residue.

Arguments:

residue	Value to initialize the CRC remainder.
---------	--

4.9 Miscellaneous Functions

4.9.1 Functions

This section describes several miscellaneous intrinsic functions. These functions are summarized in [Table 23](#).

Table 23. Miscellaneous Functions Summary (Sheet 1 of 3)

Name (args)	Description
<code>__int64 __timestamp_start(void);</code> <code>__int64 __timestamp_stop(__int64 handle);</code>	These functions use <code>local_csr</code> <code>timestamp_low</code> and <code>timestamp_high</code> to measure time elapsed in 16-cycle intervals between start and stop.
<code>int __assign_relative_register(void *x, int reg_num);</code>	Takes the address of a signal variable or transfer register variable and binds it to a physical register number.
<code>int __signal_number(SIGNAL *s, ...)</code>	Takes the address of a signal variable and returns the number of the physical signal register.
<code>int __xfer_reg_number(void *x, ...);</code>	Takes the address of a transfer register variable and returns the number of the transfer register allocated to that variable.
<code>int bit_test(unsigned int data, unsigned int bit_pos);</code>	Tests the <code>bit_pos</code> in <code>data</code> and return a 1 if set or 0 if clear.
<code>int inp_state_test(inp_state_t state);</code>	Tests the value of the specified state name and returns a 1 if the state is set or 0 if clear.
<code>int LoadTimeConstant(char *name);</code>	Causes a string constant (<code>name</code>) to be associated with an integer value at load time (using <code>ucld</code>) and returns the integer.
<code>unsigned int __ctx(void);</code>	Returns the value of the currently executing context in the range of 0 through 7.
<code>unsigned int __ME(void);</code>	Returns the value of the currently executing microengine, the range is 0x00 to 0x17.
<code>unsigned int __n_ctx(void);</code>	Returns the number of context compiled to run, the range is 0 to 7.
<code>unsigned int __nctx_mode(void);</code>	Returns the context mode 4 or 8.
<code>unsigned int __profile_count_start(void);</code> <code>unsigned int __profile_count_stop(unsigned int handle);</code>	These functions use <code>local_csr</code> <code>profile_count</code> to measure time elapse in cycle between start and stop.
<code>unsigned int __sleep(unsigned int cycles);</code>	This function suspend current thread for specified cycles. The granularity is 16.
<code>unsigned int dbl_shl(</code> <code> unsigned int srcA,</code> <code> unsigned int srcB,</code> <code> unsigned int shift_cnt);</code>	Returns the lower 32 bits of the result of concatenating <code>srcA</code> (high order bits) and <code>srcB</code> (low order bits) together and left shifting the 64 bit quantity by a specified shift count.
<code>unsigned int dbl_shr(</code> <code> unsigned int srcA,</code> <code> unsigned int srcB,</code> <code> unsigned int shift_cnt);</code>	Returns the lower 32 bits of the result of concatenating <code>srcA</code> (high order bits) and <code>srcB</code> (low order bits) together and right shifting the 64 bit quantity by a specified shift count.
<code>unsigned int ffs(unsigned int data);</code>	Finds the first (least significant) bit set in <code>data</code> and returns its bit posit. If no bits are set, the return value is undefined. Otherwise, the return value is in the range of 0 through 31.

Table 23. Miscellaneous Functions Summary (Continued) (Sheet 2 of 3)

Name (args)	Description
unsigned int multiply_16x16(unsigned int x, unsigned int y);	Returns the result of 16-bit x multiplied by 16-bit y
unsigned int multiply_24x8(unsigned int x, unsigned int y);	Returns the result of 24-bit x multiplied by 8-bit y
unsigned int multiply_32x32_hi(unsigned int x, unsigned int y);	Returns the higher 32-bit result of 32-bit x multiplied by 32-bit y
unsigned int multiply_32x32_lo(unsigned int x, unsigned int y);	Returns the lower 32-bit result of 32-bit x multiplied by 32-bit y
unsigned int nn_ring_dequeue();	Returns the next neighbor ring indexed by NN_GET without post-incrementing NN_GET.
unsigned int nn_ring_dequeue_incr();	Returns the next neighbor ring indexed by NN_GET and post-increments NN_GET.
unsigned int pop_count(unsigned int data);	(IXP28xx Rev. B and above only.) Returns the number of "1" bits in the specified value "data".
unsigned long long multiply_32x32(unsigned int x, unsigned int y);	Returns the higher 64-bit result of 32-bit x multiplied by 32-bit y
void __critical_path(<int i>)	Marks a section of coded as being on the critical path of the application. Takes an optional integer argument (0-100) that specifies the priority of overlapping critical paths. Higher numbered paths receive higher priority for code layout. The default is 100 if no argument is specified.
void __free_write_buffer(void *data);	Indicates that any pending write that writes out the data buffer can be considered as having completed. The transfer registers allocated for the write can now be used for other write operations. Same as __implicit_read().
void __implicit_read(void *x, ...);	Takes the address of a signal or transfer register variable and indicates that the signal or transfer register is being read asynchronously or implicitly by the hardware. Same as __free_write_buffer().
void __implicit_write(void *x, ...);	Takes the address of a signal or transfer register variable and indicates that the signal or transfer register is being written asynchronously or implicitly by the hardware.
void __impossible_path() (void);	Asks the compiler to perform the default case removal optimization described in Section 3.11.1 .
void __no_spill_begin();	Marks the beginning of a "no-spill" program region, where the compiler attempts to keep all the used variables in registers unless they have been explicitly allocated to memory or have had their address taken.
void __no_spill_end();	Marks the end of a "no-spill" program region.
void __no_swap_begin(void);	Starts a context swap-free critical section. See Section 3.12 .
void __no_swap_end(void);	Ends a context swap-free critical section. See Section 3.12 .
void __set_profile_count(unsigned int profile_count);	Sets local_csr_profile_count.

Table 23. Miscellaneous Functions Summary (Continued) (Sheet 3 of 3)

Name (args)	Description
void __set_timestamp(__int64 timestamp);	Sets both local_csr_timestamp_low and local_csr_timestamp_high.
void __switch_pack(enum swpack_t pack_info);	Asks the compiler to perform the switch packing optimizing described in Section 3.11.2 .
void assert(int exp);	Triggers a breakpoint on the currently executing thread.
void byte_align_block_be(unsigned int n_byte_align_oper, void *dest, void *src, unsigned shift_cnt);	Sets local_csr BYTE_INDEX to shift_cnt, then performs n_byte_align_oper times of consecutive byte_align_be operations on a pair of 32-bit element in dest and src.
void byte_align_block_le(unsigned int n_byte_align_oper, void *dest, void *src, unsigned shift_cnt);	Sets local_csr BYTE_INDEX to shift_cnt, then performs n_byte_align_oper times of consecutive byte_align_le operations on a pair of 32-bit elements in dest and src.
void global_label(int *label);	Creates a named global label at the point the intrinsic is defined.
void nn_ring_enqueue_incr(unsigned int val);	Sets the next neighbor ring indexed by NN_PUT with val and post-increments NN_PUT.

4.9.1.1 dbl_shr()

Function Syntax:

```
unsigned int dbl_shr(  
    unsigned int srcA,  
    unsigned int srcB,  
    unsigned int shift_cnt);
```

Description:

This function returns the lower 32 bits of the result of concatenating srcA (high order bits) and srcB (low order bits) together and right shifting the 64 bit quantity by a specified shift count. The shift_cnt argument must be in the range 1 through 31.

Arguments:

srcA	High order bits.
srcB	Low order bits.
shift_cnt	Number of bit positions to right shift.

4.9.1.2 dbi_shl()

Function Syntax:

```
unsigned int dbi_shl(  
    unsigned int srcA,  
    unsigned int srcB,  
    unsigned int shift_cnt);
```

Description:

This function returns the upper 32 bits of the result of concatenating srcA (high order bits) and srcB (low order bits) together and left shifting the 64 bit quantity by a specified shift count. The shift_cnt argument must be in the range 1 through 31.

Arguments:

srcA	High order bits.
srcB	Low order bits.
shift_cnt	Number of bit positions to left shift.

4.9.1.3 `__ctx()`

Function Syntax:

```
unsigned int __ctx();
```

Description:

This function returns the value of the currently executing context. The range is between 0 and 7.

Arguments:

None.



4.9.1.4 **__ME()**

Function Syntax:

```
unsigned int __ME();
```

Description:

This function returns the value of the currently executing microengine, the range is 0x00 to 0x17.

Arguments:

None.

4.9.1.5 `__n_ctx()`

Function Syntax:

```
unsigned int __n_ctx();
```

Description:

This function returns the number of contexts compiled to run. The range is 0 to 7.

Arguments:

None.



4.9.1.6 **`__nctx_mode()`**

Function Syntax:

```
unsigned int __nctx_mode();
```

Description:

This function returns the context mode, either 4 or 8.

Arguments:

None.

4.9.1.7 **sleep()**

Function Syntax:

```
unsigned int sleep(unsigned int cycles);
```

Description:

This function suspends the current thread for the specified number of cycles. The granularity is 16.

Arguments:

cycles	Number of cycles to suspend the current thread.
--------	---



4.9.1.8 ffs()

Function Syntax:

```
unsigned int ffs(unsigned int data);
```

Description:

This function finds the first (least significant) bit set in data and returns its bit position. If there are no bits set (i.e., the data argument is 0) then the return value is undefined. Otherwise, the return value is in the range 0 through 31.

Arguments:

data	Data to examine.
------	------------------

4.9.1.9 `__LoadTimeConstant()`

Function Syntax:

```
int __LoadTimeConstant(char *name);
```

Description:

This function associates a constant name with an integer value at load time using `uclid` and returns the integer. This provides equivalent functionality as the `.import_var` using assembler. There is no register or memory allocated to this constant. The name argument uniquely identifies the constant.

Arguments:

name	The name of a constant to bound at load time.
------	---

Example:

In the following example, A1 and A2 have the same constant, which may or may not be the same as the independent constant for B.

```
A1 = LoadTimeConstant("CONST_A");
```

```
A2 = LoadTimeConstant("CONST_A");
```

```
B = LoadTimeConstant("CONST_B");
```

4.9.1.10 `__global_label()`

Function Syntax:

```
void __global_label(int *label);
```

Description:

This function creates a named global label at the point the intrinsic is defined.

Arguments:

label	Name of the label to create.
-------	------------------------------

Examples:

1. This example creates a label named `ixp_start_packet_count` at the intrinsic invocation point. The name of the label is exactly as specified (i.e., no “_” is prepended to the name). The name of the label must be unique. The label does not interact with other C labels since C labels are renamed.

```
global_label("ixp_start_packet_count");
```

2. This fragment creates two different labels -- one global LABEL# and some other option/code specific label, such as `l_345#`.

```
global_label(LABEL);  
...  
LABEL:
```

4.9.1.11 multiply_24x8()

Function Syntax:

```
unsigned int multiply_24x8(unsigned int x, unsigned int y);
```

Description:

This function returns the result of 24-bit x multiplied by 8-bit y.

Arguments:

x	24-bit int to multiply.
y	8-bit int to multiply.



4.9.1.12 multiply_16x16()

Function Syntax:

```
unsigned int multiply_16x16(unsigned int x, unsigned int y);
```

Description:

This function returns the result of 16-bit x multiplied by 16-bit y.

Arguments:

x	16-bit int to multiply.
y	16-bit int to multiply.

4.9.1.13 multiply_32x32_lo()

Function Syntax:

```
unsigned int multiply_32x32_lo(unsigned int x, unsigned int y);
```

Description:

This function returns the lower 32-bit result of 32-bit x multiplied by 32-bit y.

Arguments:

x	32-bit int to multiply.
y	32-bit int to multiply.



4.9.1.14 multiply_32x32_hi()

Function Syntax:

```
unsigned int multiply_32x32_hi(unsigned int x, unsigned int y);
```

Description:

This function returns the higher 32-bit result of 32-bit x multiplied by 32-bit y.

Arguments:

x	32-bit int to multiply.
y	32-bit int to multiply.

4.9.1.15 multiply_32x32()

Function Syntax:

```
unsigned long long multiply_32x32(unsigned int x, unsigned int y);
```

Description:

This function returns the higher 32-bit result of 32-bit x multiplied by 32-bit y.

Arguments:

x	32-bit int to multiply.
y	32-bit int to multiply.



4.9.1.16 `__set_timestamp()`

Function Syntax:

```
void __set_timestamp(__int64 timestamp);
```

Description:

This function sets both the `local_csr_timestamp_low` and `local_csr_timestamp_high` fields of the `local_csr_t` enum.

Arguments:

<code>timestamp</code>	Timestamp to set.
------------------------	-------------------

4.9.1.17 `__timestamp_start()`, `__timestamp_stop`

Function Syntax:

```
__int64 __timestamp_start(void);  
__int64 __timestamp_stop(__int64 handle);
```

Description:

These functions use `local_csr_timestamp_low` and `timestamp_high` to measure time elapse in 16-cycle intervals between start and stop. The `__timestamp_start()` function returns a handle that is used by the `__timestamp_stop()` function.

Arguments:

handle	Timestamp handle returned by the <code>__timestamp_start()</code> function and passed to the <code>__timestamp_stop()</code> function.
--------	--

4.9.1.18 `__set_profile_count()`

Function Syntax:

```
void __set_profile_count(unsigned int profile_count);
```

Description:

This function sets the `local_csr_profile_count` field of the `local_csr_t` enum.

Arguments:

<code>profile_count</code>	Profile count to set.
----------------------------	-----------------------

4.9.1.19 `__profile_count_start()`, `__profile_count_stop`

Function Syntax:

```
unsigned int __profile_count_start(void);  
unsigned int __profile_count_stop(unsigned int handle);
```

Description:

These functions use `local_csr_profile_count` to measure time elapse in cycle between start and stop. The `__profile_count_start()` function returns a handle that is used by the `__profile_count_stop()` function.

Arguments:

handle	Handle returned by the <code>__profile_count_start()</code> function that is passed to the <code>__profile_count_stop()</code> function.
--------	--



4.9.1.20 `__signal_number()`

Function Syntax:

```
int __signal_number(SIGNAL* s, ...);
```

Description:

This function takes address of a signal variable and returns the number of the physical signal register allocated to that variable. This is useful for operations such as setting up of the RX_THREAD_FREELIST CSRs. For signals declared as remote, the `__signal_number()` takes two arguments, the first being the address of the signal and the second being the microengine number in which the remote signal resides.

Arguments:

s	Pointer to a signal variable.
...	If the signal is declared as remote, the second argument should be an unsigned integer of the microengine on which it resides.

4.9.1.21 `__xfer_reg_number()`

Function Syntax:

```
int __xfer_reg_number(void* x, ...);
```

Description:

This function takes the address of a transfer register variable and returns the number of the transfer register allocated to that variable. This is useful for operations such as setting up of the `RX_THREAD_FREELIST` CSRs. For transfer registers declared as remote, the `__xfer_reg_number()` intrinsic takes two arguments, the first being the address of the transfer register and the second being the microengine number in which the transfer register resides.

Arguments:

x	Pointer to a transfer register.
...	If the signal is declared as remote, the second argument should be an unsigned integer of the microengine on which it resides.

4.9.1.22 `__assign_relative_register()`

Function Syntax:

```
int __assign_relative_register(void *x, int reg_num);
```

Description:

This function takes the address of a signal variable or transfer register variable, and binds it to a physical register number. If the address of a structure or array is passed, the first element of the structure or array will be assigned the physical register number, and every successive element will be assigned a consecutive register number.

Arguments:

x	Address of a signal variable or transfer register variable.
reg_num	Physical register number to bind variable to.

4.9.1.23 `__implicit_read()`

Function Syntax:

```
void __implicit_read(void *x, ...);
```

Description:

This function takes as argument the address of a signal or transfer register variable and indicates that the signal or transfer register is being read asynchronously or implicitly by the hardware. It is necessary to use this intrinsic to mark all definitions and use points of signal/transfer registers that are not directly visible to the compiler. They ensure that the compiler does correct lifetime analysis and hence correct register allocation to such variables. This intrinsic must be used, for example, when the `RX_THREAD_FREELIST` CSR is written with the register number allocated to a variable, when a signal is requested by writing into a CSR, or a signal is tested by doing a `ctx_arb` with a signal mask generated with `__signals()`.

Note: This intrinsic performs the same task as `__free_write_buffer()`.

Arguments:

<code>x</code>	Pointer to a signal or transfer register address.
<code><count></code>	Optional integer argument that specifies the size in bytes of the read. This can be used to target specific elements of a structure or array.

4.9.1.24 `__implicit_write()`

Function Syntax:

```
void __implicit_write (void *x, ...);
```

Description:

This function takes as argument the address of a signal or transfer register variable and indicates that the signal or transfer register is being written asynchronously or implicitly by the hardware. It is necessary to use this intrinsic to mark all definitions and use points of signal/transfer registers that are not directly visible to the compiler. They ensure that the compiler does correct lifetime analysis and hence correct register allocation to such variables. This intrinsic must be used, for example, when the `RX_THREAD_FREELIST` CSR is written with the register number allocated to a variable, when a signal is requested by writing into a CSR, or a signal is tested by doing a `ctx_arb` with a signal mask generated with `__signals()`. See [Section 3.8, “User Assisted Live Range Analysis” on page 55](#) and [Section 7.2, “Things to Remember When Writing Microengine C Code” on page 374](#) for a more complete explanation with examples.

Arguments:

<code>x</code>	Pointer to a signal or transfer register address.
<code><count></code>	Optional integer argument that specifies the size in bytes of the write. This can be used to target specific elements of a structure or array.

4.9.1.25 `__free_write_buffer()`

Function Syntax:

```
void __free_write_buffer(void *data);
```

Description:

This intrinsic is called after an asynchronous memory write operation has been issued. It indicates that all pending writes that require the given data buffer have completed. The transfer registers allocated for the write can then be reused for other write operations. Without a call to this intrinsic, the compiler will assume that transfer registers involved in an asynchronous memory write can be reused immediately after the operation has issued, which may cause invalid data to be written out to memory. See [Section 3.8, “User Assisted Live Range Analysis” on page 55](#) and [Section 7.2, “Things to Remember When Writing Microengine C Code” on page 374](#) for a more complete explanation with examples.

Note: This intrinsic performs the same task as `__implicit_read()`.

Arguments:

<code>data</code>	Data buffer to write.
-------------------	-----------------------



4.9.1.26 `inp_state_test()`

Function Syntax:

```
int inp_state_test(inp_state_t state);
```

Description:

This function tests the value of the specified state name and returns a 1 if the state is set or 0 if clear. Argument state must be a constant literal as required by the microcode assembler; otherwise, the compiler generates a runtime check, if possible, with loss of performance

Arguments:

state	State to test.
-------	----------------

4.9.1.27 **bit_test()**

Function Syntax:

```
int bit_test(unsigned int data, unsigned int bit_pos);
```

Description:

This function tests the bit_pos in data and return a 1 if set or 0 if clear.

Arguments:

data	Integer to test.
bit_pos	Bit position in data to test.



4.9.1.28 nn_ring_dequeue_incr()

Function Syntax:

```
unsigned int nn_ring_dequeue_incr();
```

Description:

This function returns the next neighbor ring indexed by NN_GET, then post-increments NN_GET.

Arguments:

None.

4.9.1.29 nn_ring_dequeue()

Function Syntax:

```
unsigned int nn_ring_dequeue();
```

Description:

This function returns the next neighbor ring indexed by NN_GET without post-incrementing NN_GET.

Arguments:

None.



4.9.1.30 `nn_ring_enqueue_incr()`

Function Syntax:

```
void nn_ring_enqueue_incr(unsigned int val);
```

Description:

This function sets the next neighbor ring indexed by NN_PUT with val and post-increments NN_PUT.

Arguments:

val	Value to set next neighbor ring indexed by NN_PUT.
-----	--

4.9.1.31 byte_align_block_le()

Function Syntax:

```
void byte_align_block_le(  
    unsigned int n_byte_align_oper,  
    void *dest,  
    void *src,  
    unsigned shift_cnt);
```

Description:

This function sets local_csr BYTE_INDEX to shift_cnt, then performs n_byte_align_oper times of consecutive byte_align_le operations on a pair of 32-bit elements in dest and src. Arguments dest and src are addresses of Xfer/GPR 32-bit variables (or aggregates of 32-bit elements) that must be enregisterized.

Arguments:

n_byte_align_oper	The number of byte_align operations to perform on dest and src.
dest	Address that stores the results of the alignment shift.
src	Address that contains the pair of 32-bit elements to shift.
shift_cnt	The number of bytes to shift.

4.9.1.32 `byte_align_block_be()`

Function Syntax:

```
void byte_align_block_be(  
    unsigned int n_byte_align_oper,  
    void *dest,  
    void *src,  
    unsigned shift_cnt);
```

Description:

This function sets local_csr BYTE_INDEX to shift_cnt, then performs n_byte_align_oper times of consecutive byte_align_be operations on a pair of 32-bit elements in dest and src. Arguments dest and src are addresses of Xfer/GPR 32-bit variables (or aggregates of 32-bit elements) that must be enregisterized.

Arguments:

n_byte_align_oper	The number of byte_align operations to perform on dest and src.
dest	Address that stores the results of the alignment shift.
src	Address that contains the pair of 32-bit elements to shift.
shift_cnt	The number of bytes to shift.

4.9.1.33 `__no_spill_begin()`

Function Syntax:

```
void __no_spill_begin();
```

Description:

Marks the beginning of a "no-spill" program region, where the compiler attempts to keep all the used variables in registers, unless they have been explicitly allocated to memory or have had their address taken. This is done at the expense of other program regions, which may incur extra spills. If the compiler cannot allocate all the variables to registers, compilation will halt with an error message.

Arguments:

None.



4.9.1.34 `__no_spill_end()`

Function Syntax:

```
void __no_spill_end();
```

Description:

Marks the end of a "no-spill" program region.

Arguments:

None.

4.9.1.35 assert()

Function Syntax:

```
void assert(int exp);
```

Description:

If `exp` is zero, a message is printed to standard output (see the note below) and the `ctx_arb[bpt]` instruction is executed, which triggers a breakpoint on the currently executing thread.

Note: This function requires Transactor console I/O support, which is implemented in the Transactor script file “MicroengineC\samplesutil\util.ind,” under the SDK install directory. Add this script to your project with “Project\Insert Script Files...” and set it to execute on startup with “Simulation\Options...\Startup.” If you are using command line execution, the .ind script generated by the compiler already contains Transactor console I/O support. Microengine C does not currently support hardware console I/O - on IXP hardware; the `ctx_arb[bpt]` instruction will be executed if an assertion fails, but no output will be printed.

Arguments:

<code>exp</code>	Expression to test.
------------------	---------------------

4.9.1.36 `__critical_path()`

Function Syntax:

```
void __critical_path(<int i>); // "i" is optional
```

Description:

Marks a section of code as being on the critical path of the application. Takes an optional integer argument (0-100) that specifies the priority of overlapping critical paths. Higher numbered paths receive higher priority for code layout. The default is 100 if no argument is specified. [Section 3.10, “Critical Path Annotation and Code Layout”](#) on page 59 for details.

Arguments:

i	Optional argument; indicates priority from 0-100. Default is 100
---	--

4.9.1.37 pop_count()

Function Syntax:

```
unsigned int pop_count(unsigned int data);
```

Description:

(IXP28xx Rev. B and above only) This function returns the number of “1” bits in the given value “data”.

Arguments:

data	The value on which to perform the operation.
------	--

4.9.1.38 `__switch_pack`

Function Syntax:

```
void __switch_pack(enum swpack_t pack_info);
```

Description:

Asks the compiler to perform the switch block packing optimization described in [Section 3.11.2](#). This function should only be placed in the “default” handler of a switch() argument.

Arguments:

<code>pack_info.</code>	The type of the switch packing optimization desired, described in Section 3.11 .
-------------------------	--

4.9.1.39 `__impossible_path`

Function Syntax:

```
void __impossible_path() (void);
```

Description:

Asks the compiler to perform the default case removal optimization described in [Section 3.11.1](#). This function should only be placed in the “default” handler of a `switch()` statement.

Arguments:

none



4.9.1.40 `__no_swap_begin`

Function Syntax:

```
void __no_swap_begin(void);
```

Description: Starts a context swap-free critical section. See [Section 3.12](#) for more details

Arguments:

none

4.9.1.41 `__no_swap_end`

Function Syntax:

```
void __no_swap_end(void);
```

Description: Ends a context swap-free critical section. See [Section 3.12](#) or more details

Arguments:

none

4.10 Restrictions On Ininsics

4.10.1 Intrinsic Function Arguments that Map to Transfer Registers in Microcode

The memory and csr intrinsic functions each take an argument that points to a buffer of memory. This buffer is mapped to transfer registers and due to the read-only/write-only restrictions on transfer registers, and due to their asynchronous update, the compiler restricts their usage. In the following, *xfer buffer address* refers to the buffer pointer that is passed to these intrinsics as an argument whereas *xfer buffer* refers to the memory locations referred to by the *xfer buffer address*. For example, in the following code:

```
{
    __declspec(sram) long long x;
    __declspec(sram_read_reg) long long rd, buf[1];
    SIGNAL sig;

    sram_read(&rd, &x, 2, ctx_swap, &sig);
    sram_read(buf, &x, 2, ctx_swap, &sig);
}
```

rd, and buf[] are the *xfer buffers* and &rd, and buf are the *xfer buffer addresses*.

__declspec(sram_read_reg) and other transfer register data types are used to make you aware of the restrictions listed below.

The following are the restrictions imposed and are illustrated with examples below:

1. The *xfer buffer* argument of a read intrinsic cannot be redefined by any other instruction. However, it is possible to re-use it in another read intrinsic to the compatible memory region if the reads are not asynchronous, or if their lifetimes do not overlap.
2. The *xfer buffer* argument of a write intrinsic cannot be read by any other instruction. However, it is possible to re-use it in another write intrinsic to the compatible memory region if the writes are not asynchronous, or if their lifetimes do not overlap.
3. The *xfer buffer* argument to a read intrinsic cannot be used as the *xfer buffer* argument to a write intrinsic and vice versa.
4. It is in general not permitted to assign or take the address of an xfer buffer except when passing it to the intrinsic. See [Section 7.2, “Things to Remember When Writing Microengine C Code”](#) for more guideline.
5. The *xfer buffer address* argument to an intrinsic must be supplied by a direct reference to a buffer variable, and not through a pointer variable.
6. An *xfer buffer* cannot be part of a larger declared aggregate in memory.
7. For intrinsics that do an asynchronous write (i.e. not ctx_swap), you must mark the spot in the code where the operation has been tested and is known to be complete. This is done by calling the intrinsic function free_write_buffer(&x) where x is the *xfer buffer* used in the write.
8. Intrinsics that perform asynchronous reads may require the use of __implicit_read(). See [Section 7.2, “Things to Remember When Writing Microengine C Code”](#) on page 374 for details.
9. Parameter count to intrinsics are preferred to be constant. The compiler may generate an indirect_ref with performance loss if it can't be resolved to a constant at compile-time, or it will error if it's not possible. Parameters sync and csr must be resolved to be constant at compile-time.

Example:

The following example shows violations of these restrictions:

```

{
    __declspec (sram) long long x;
    __declspec (sram) int y;
    int *rdp;
    SIGNAL s;
    __declspec(sram_read_reg) int rd;
    __declspec(sram_write_reg) int wr;
    __declspec(sram_read_reg) int b[2];
    int buf[100];
    int mask;
    ...
    sram_read(b, &x, 2, sig_done, &s);
    ...
    y=b[i];           // Violates restr 4. Address (b) implicitly
                    // taken since i is not a constant.
    rdp = buf;       // Violates restr 4.
    sram_read(&rd, &y, 1, sig_done, &s);
    sram_read(rdp, &x, 2, sig_done, &s); // Violates restr 5.
    sram_write(&wr, &y, 1, sig_done, &s);
    sram_write(buf, &x, 2, sig_done, &s); // Violates restr 6.
    ...
    rd += 1;        // Violates restr 1.
    if (wr)         // Violates restr 2.
    {
        rdp = &rd; // Violates restr 4.
    }
    sram_write(&rd, &y, 1, sig_done, &s); // Violates restr 3.
}

```



Inline Assembly Language

5

The compiler supports inline assembly language through the `C__asm` statements and blocks, providing full access to the compiler microengine instruction set. Inline assembler instructions must refer symbolically to variables declared in C.

All references must be symbolic—the compiler then assigns the registers. It does not allow references to specific machine registers.

The preprocessor and assembler directives supported for the microcode assembler are not supported by the C compiler.

Note: The compiler checks for hazards in the inline assembly sequence and flags any hazards found as errors. The philosophy in the compiler is to not interfere with inline assembly code in a way that would make the code worse from a performance point of view. The assumption is that users who write inline assembly are sophisticated in programming at the microcode level and generally do not want the compiler to modify or rearrange inline assembly. The only exception to this is that the compiler may perform some low level optimizations and will try to fill delay slots.



5.1 Single `__asm` Instruction

For a single instruction, the following form is accepted:

```
__asm <instruction>;
```

5.2 Block of `__asm` Assembly Code

For a block of assembly code, the following is accepted:

```
__asm __attribute(CONSTANT)
{
label:    <instruction>; comment
        .
        .
}
```

The allowed values for the “CONSTANT” block attribute are `ASM_HAS_JUMP` and `LITERAL_ASM`. The `LITERAL_ASM` attribute disables all compiler optimizations in the assembly block and allows the use of the `defer[]` keyword. The `ASM_HAS_JUMP` attribute is used when the assembly block contains the `jump[]` instruction and is further described below. Assembly block attributes can be combined with the OR (`|`) operator.

Note: The label, comment, and attribute value are optional. The attribute value, if specified, must be a constant. The semi-colon begins a comment in the asm block that continues up to and including the end-of-line character.

5.3 Instruction Format

An inline instruction has the following format:

```
opcode [operand, operand...], keyword, ...
```

The allowed operands and optional keywords depend on the opcode.

Function calls are supported using a pseudo instruction CALL that takes a single argument of the function name:

```
CALL[foo]
```

The compiler expands this into storing the return address and branching to the function.

Inline assembly blocks containing jump[] instructions need to be marked with the attribute constant ASM_HAS_JUMP (defined in ixp.h). The jump[] instructions themselves also need to contain a list of possible targets. The maximum number of jump targets is 120.

For example:

```
__asm __attribute(ASM_HAS_JUMP)
{
    alu_shf[offset, K, +, I, <<1]; // example offset calculation
    jump[offset, base], targets[L1, L2, L3] // jump to L1, L2, or L3
    L1:
    ...
    L2:
    ...
    L3:
    ...
}
```

This added information allows the compiler to properly allocate registers and optimize code within the assembly block.

5.4 Operand Syntax

Each operand to the instruction specifies one of the following:

- a register
- an immediate value (constant)
- a label
- a keyword, such as memory or ALU operation

The specific type of each operand is determined by the opcode.

5.4.1 Register Operands

When a register operand is called for, you must supply a special hardware register, such as a CSR name (where this is applicable) or a C variable that is in a register, that is either a local, an argument, or a global variable without a memory attribute (sram, dram, scratch, or local memory).

The register variable must be an integral type (no char or short or `__int64`). After setting up the local memory address CSRs, you can use `*I$index0` or `*I$index1` as register operands to address local memory. If the variable specifies a struct or union type, you can use the `."` notation to select a field. The field must be a full 32-bit item. Fields of type char or short, and bit fields cannot be referenced in this manner. The variable name cannot be the same as a reserved token in the microcode language. This includes, for example, all the tokens that arise from the ALU opcode such as B, A, CARRY, AND, OR, XOR, IFSIGN or common names like `csr` and `inp_state` etc. These tokens are case insensitive and hence neither variable `"a"` nor `"A"` is permitted as a register operand. Note that B, and A are reserved tokens since `"B-A"` is an ALU opcode and white space is permitted between `"B"`, `"-"`, and `"A"`. Other examples of reserved tokens are (but not limited to) `"csr"`, `"state"`, and `"inp_state"`.

Objects in registers can be addressed with a byte offset by adding a `"+n"` to the object name. For example:

```
__declspec(gp_reg) int thing[10];
__asm {
    alu[thing+12, --, B, 4]; // thing[3] = 4
}
```

If the object is assigned to general-purpose registers, any byte offset can be used, regardless of alignment. The compiler will generate the proper mask-and-shift operations to perform an unaligned access, if necessary. Objects in transfer registers or remote next neighbor registers cannot be accessed with unaligned offsets.

When setting up an argument for a function call, you can refer to the registers used to pass the argument using the following notation:

```
funcname.arg_n
```

where `n = 0` for the first (leftmost) argument in the function header.

To get the return value from a function after a call, use the following:

```
funcname.ret
```

If an argument or return value is a struct, you can cast it to the struct type and select a field as follows:

```
funcname.arg_n:structname.field
```

The example in [Section 5.4.3](#) depicts usage of C variables, structure field access, and function call setup within inline assembly.

When calling functions inside inline assembly blocks, you must take care to pass the arguments in the way that the function expects. The compiler will treat all such arguments as untyped register values and will not perform type checking.

For operands that span more than one register, you can use an offset of 4 bytes for each additional register, for example:

```
foo+8
```

refers to the third register occupied by foo.

5.4.2 Immediate Operands

Immediate operands are constant expressions using C syntax. All of the C operators are supported, including the sizeof() macro. An “offsetof(struct, field)” macro is also supported, which returns the byte offset of a given field in a struct.

Simple “constant folding” is supported. For example, “3+4” will be replaced with the constant “7”.

The address of a memory variable can also be specified as an immediate operand, using the & operator. Since immediate operands can be no more than 16 bits, you can get the high order 16 bits of an address by using the >> operator as follows:

```
&buffer >> 16
```

Note: See the *IXP2800 Network Processor Programmer’s Reference Manual* for allowable opcodes, operands, and keywords.

Note: A non-zero constant beginning with zero (0) is interpreted as an octal number. Thus, for example, the byte mask in `__asm {ld_field[dest, 0011, src, >>16]}` will be interpreted as 9. To express the mask in binary use the “b” or “B” suffix, as in `__asm {ld_field[dest, 0011b, src, >>16]}`. To express the mask in hex, use 0x or OX as a prefix, as in `__asm {ld_filed[dest, 0x11, src, >>16]}`.

5.4.3 Usage Examples

Example: This example illustrates how to call a C function from inline assembler code and access structure fields.

```
typedef struct
{
    int a;
    int b;
} a_struct;

int a_func(int x, a_struct str)
```

```
{
    return x + str.a + str.b;
}

int call_a_func(int y)
{
    int ret;
    asm
    {
        alu    [a_func.arg_0, --, B, y]; pass y as x
        alu    [a_func.arg_1:a_struct.a, --,B,1]; pass 1 as str.a
        alu    [a_func.arg_1:a_struct.b, --,B,2]; pass 2 as str.b
        call   [a_func]
        alu    [ret, --, B, a_func.ret]; return value
    }
    return ret;
}
```



Example: This example illustrates different errors in usage.

```
__declspec(sram) int x;
int y;
__declspec(local_mem) B, C;
struct { char c; short s;} st;

foo(&y, &C);
__asm {
    alu [x, C, +, C];    Error since x has been allocated to SRAM
    alu [st.c, st.s, + st.s]; Error since s.c and s.s are of
                        type char, and short
                        respectively
    alu [B, C, +, C];    Error since B is a reserve
                        keyword in microcode
}
```

5.5 Restrictions on Use Of Assembly Language

The compiler can not handle all possible uses of assembly language. The following restrictions are in place:

- The target of a jump instruction must reside in the same inline assembly block as the branch.
- A goto statement in Microengine C code cannot branch into inline assembly code.
- Instructions that override the register address for transfer registers resulting in the registers that are not local to the current thread do not work properly.
- `funcname.arg_n` can not be used as a source operand, `funcname.ret` can not be defined. Sequences of `funcname.arg_n` definitions must be followed by inline-asm call to `funcname` without other intervening call(s) in between. All parameters to inline-asm call must be defined before call.
- The DEFER keyword can only be used in inline assembly blocks that are tagged with the LITERAL_ASM attribute.
- Code within an `__asm` block may not depend on the value of processor flags (e.g. +carry, condition codes), generated by C code outside the `__asm`.
- Register operands must be of integral type (no char, short, or `__int64`)
- The `br[]` token on the `ctx_arb[]` instruction is not supported.
- Local memory variables cannot be accessed in inline assembly using their “C names”, because the access may need to be expanded to several instructions. Local memory can be accessed directly using the local memory pointer CSRs.
- Code that depends on a specific clock cycle timing to read or write different CSR values may not function correctly. For example:

```
local_csr_write[active_lm_addr_0, a0]
local_csr_write[active_lm_addr_0,b0]
nop
nop
alu[$1, a2, +, *1$index0] //expecting to use the old value
alu[$2, a3, +, *1$index0] //expecting to use the new value
```

The compiler may insert additional NOP instructions to maintain the 3-cycle latency between the CSR writes and the CSR accesses, which would disturb the timing of the above code.

Further restrictions may be specified in the future.



Compiler Optimizations

6

The Microengine C Compiler optimizes for size, speed, or debugging. It optimizes at the function level or on a whole program level.

6.1 Machine Independent Optimizations

The compiler does several machine independent optimizations including:

- Inlining and whole program constant propagation.
- Traditional scalar optimizations to clean up the code and remove redundant computations.
- Loop unrolling/peeling—reduces taken branches and provides optimization opportunities.
- Constant and copy propagation:
Example: `y=5; x=y; foo(x);` In `foo(x) z=z<<(x+1)` is optimized as `z=z<<6`.
- Removal of dead stores—stores to memory locations that are not referenced are eliminated. These are not applied to variables declared volatile. The assignment to `x` is eliminated in the following example:

```
main() {__declspec(sram) x; x=5; return}
```

6.2 Network Processor Specific Optimizations

6.2.1 Registrations

All local and global variables are kept in registers unless:

- The address of the variable is taken and can not be propagated to the dereferencing site or used for something else than simple dereferencing.
— Example: `int x, *p; p=&x; /* p is in a register, x is in SRAM */`
- The variable is declspeced to a memory region.
— Example: `__declspec(sram) int x, *p; /* p is in a register, x is in SRAM */`
- Registers need to be spilled.
 - Transfer registers are spilled to GPRs
 - GPRs spilled to local memory or SRAM.

6.2.2 Read/Write Combining

Multiple reads or writes can be combined into one read or write respectively, if the reads/writes are to contiguous memory locations. This is done with SRAM/DRAM/SCRATCH memory operations specifying a reference count greater than one.

```

— Example:
struct{
  int a, b;
}
__declspec(sram) s;

x=s.a+s.b; //Only one SRAM read is generated with a two-word count
  
```

Although the two accesses to the SRAM structure are distinct in the C code, they are combined by the compiler into a single SRAM read instruction. One SRAM read of two words will complete faster than two separate SRAM reads, especially when the reads are blocking (ctx_swap) reads.

6.2.3 Peephole Optimization

Some instructions on the IXP architecture can perform multiple operations simultaneously. The compiler will try to use these instructions whenever possible.

Examples:

- Combining shift and logical operations.
 - `dest = x & (j << 2)` becomes `alu_shf[dest, x, AND, j, << 2]`
- Combining add and mask operations:
 - `dest = x + (j & 0xff)` becomes `alu[dest, x, +8, j]`

6.2.4 Defer Slot Filling

Some multi-cycle instructions on the IXP architecture contain “defer slots” which allow other instructions to be executed while the slower instruction is being processed. The compiler will take advantage of these defer slots when possible. For example:

```

z += 5;
if (x) goto label;
  
```

becomes:

```

alu[x, --, B, x]
beq[label#], defer[1]
alu[z, z, +, 5] ; always executed
  
```

6.2.5 Local Memory Grouping

The compiler attempts to allocate local memory variables used in the same control flow region into one continuous group and controls access to those variables through a single `local_csr_write` instruction. For information on how this is done, see [Section 3.2.7](#).

6.2.6 Local Memory Autoincrement/Autodecrement Conversion

The compiler will generate autoincrement and autodecrement addressing for local memory accesses in loops whose addresses vary by a constant amount. For example:

```
__declspec(local_mem) int arr[10];

for (i = 0; i < 10; i++)
{
    arr[i] = i;          // address of LM access increments by
                        // 1 word on every iteration
}
```

becomes:

```
local_csr_wr[active_lm_addr_1, arr]
nop
nop
nop
l_2#:
alu[*l$index1++, --, B, i]
alu[i, i, +, 1]
alu[--, i, -, 10]

blt[l_2#]
```

Previously, such accesses were performed by writing the local memory index register on each access, which would cause up to a four-cycle delay on every loop iteration.

This optimization is disabled by a switch, "-Qlm_unsafe_addr". This should be used when local memory pointers are written with invalid values and accessed conditionally. For example:

```
__declspec(local_mem) int p[100];
int i;

for (i = -100; i < 100; i++)
{
    if (i > 0)
    {
        ... = p[i];
    }
}
```

For the first 100 iterations of the loop, "p[i]" is not a valid local memory address. The above example is correct code because the loop checks if "i" has a valid value before performing the array access, but the local memory address optimizer may generate:

```
local_csr_wr[active_lm_addr_1, "&(p[-100])"]
nop
nop
nop
top:
alu[--, --, B, i] // if (i > 0)
ble[skip#]
... [..., *l$index1] // ... = p[i]
skip:
alu[--, --, B, *l$index1++] // p++ and i++
alu[i, i, +, 1]
```

```
alu[--, i, -, 100]
blt[top#]
```

The local memory index register is initialized to “&(p[-100])”, which may be negative, and incremented by 1 word every iteration. This is “functionally correct” but may not work correctly on actual IXP hardware, since signed arithmetic is not guaranteed to work on local memory index registers.

If you have code such as the above, where a local memory array or pointer expression is intentionally set to point to a value outside of the allowable 0-640 address range, you should turn off local memory postinc/postdec optimization with `-Qlm_unsafe_addr`.

6.2.7 Scheduling

The compiler may re-order instruction sequences to reduce nops; otherwise, it needs to maintain correct latencies between performance. Listed below are some cases where nops are needed.

- Between writes to `local_csr` and the use or read of the same `local_csr`, included, but not limited to the following:
 - Writes to `lm_addr` CSRs and `*l$index0` or `*l$index1`
 - Writes to `T_index` CSRs and `*$index` or `*$$index`
 - Writes to `BYTE_INDEX` CSRs and `byte_align_be` or `byte_align_le`
 - Writes to `NN_Put/NN_get` and `*n$index`
- Between instruction setting condition code and `bcc` on it with `defer[3]`
- Between `crc` operations, and between the last `crc` and the use of `crc` remainder
- Between definition of nearest neighbor and use of the same `nn` in self-mode
- Between two `local_csr` writes to the following 3 `local_csrs`: `next_neighbor_signal`, `prev_neighbor_signal`, and `same_me_signal`

The scheduler assumes that you write serialized code without knowledge of pipeline latencies, which might change from generation to generation, which results in non-portable code. The Hazard Detector inserts proper nops in `-Od` when the scheduler is turned off.

6.2.8 I/O Parallelization

Multiple I/O operations with the `ctx_swap` token can be converted to use `sig_done` followed by a `ctx_arb` to wait a mask of their signals, if instructions between I/O and the inserted `ctx_arb`, if any, don't access other global states, and if doing so won't create more spilling. The compiler may use an existing `ctx_arb` rather than inserting a new one if one is found and the above rules are satisfied. If I/O with the `ctx_swap` token is generated by the compiler from C statements (as opposed to intrinsics or inline-assembly), meaning user has no expectation of `ctx_swap`, the restriction about accessing global states may be relaxed. For example:

Original sequence of code:

```
sram[read, sr, &mem1, 0, 1], ctx_swap[sig1]
sram[write, sw, &mem2, 0, 4], ctx_swap[sig2]
dram[read, dr, &mem3, 0, 2], sig_done[sig3]
ctx_arb[sig3]
```

Can be converted into:

```
sram[read, sr, &mem1, 0, 1], sig_done[sig1]
sram[write, sw, &mem2, 0, 4], sig_done[sig2]
dram[read, dr, &mem3, 0, 2], sig_done[sig3]
ctx_arb[sig1 sig2 sig3]
__free_write_buffer(&sw);
```

The SRAM read and write operations can be executed in parallel with the DRAM read. This is possible because the code between the SRAM read and the `ctx_arb` instruction meets the conditions described above. Namely, the transfer register “sr” is not read, there are no global states accessed, and no spills will be caused by extending the live ranges of the transfer registers and signals.



Tips for Optimization, Troubleshooting, and Debugging 7

The following are suggestions on how you can optimize, troubleshoot, and debug your code.

7.1 Optimizing Your Code

The C compiler performs automatic optimization at various levels but there are some coding techniques that will make your program perform better.

- Minimize variable allocation to memory.
 - Taking the address of a variable or declaring it with a memory region attribute causes allocation to memory.
 - Structures/arrays larger than 64 bytes (or 128 bytes in 4-context mode) are allocated to memory.
- Minimize access to memory variables.
- When possible, it is preferable to declare a variable that does not require initialization as a local variable in `main()` rather than as a global or a static variable. This is because a global/static declaration implies a default initialization to zero. Such variables when allocated to registers or local memory are initialized at runtime. By making it a local variable the unnecessary initialization is avoided.
- When using the `ctx()` intrinsic, it is preferable to use it directly in a branch condition. This is because the architecture supports the `BR=CTX` and `BR!=CTX` instructions.

Example:

```
if (ctx() == 0) { ... }
else if (ctx() == 1) { ... }
...
```

or the alternative:

```
switch (ctx())
case 0:
    ...
    break;
case 1:
    ...
    break;
```

are both preferable to:

```
unsigned int c = ctx();
if (c == 0) { ... }
else if (c == 1) { ... }
```

Use unsigned types where signed types are not needed.

- Arithmetic right shift is more expensive than logical right shift.

When writing into bit fields, do not mask unnecessarily.

Example:

```
sram_read_write_ind_t ind;  
ind.xadd_reg & 0x1f; /* mask is not needed as xadd is 5 bits */
```

Force inlining of very short (1 or 2 microword) functions with `__forceinline` directive.

- The UC assembler preprocessor supports the use of loops to evaluate constant expressions at assembly time. The use of such loops is not recommended in Microengine C code. The compiler will "fold" simple expressions at compile time (example: "i = 4 + 5" will be folded to "i = 9"), but loops will not be pre-evaluated in this manner.
- Whenever possible, use `__declspec(aligned(n))` to inform the compiler about guaranteed run time characteristics of the pointer variable. This information allows the compiler to generate significantly better code. It is your responsibility to specify correct values for alignment boundaries. Incorrect alignment information might force the compiler to generate incorrect code.
- Use the "restrict" qualifier on pointers when there are no other means to access the memory it points to. Please refer to [Section 3.7.4.1](#), "The "restrict" Qualifier" on page 54 for details.

7.2 Things to Remember When Writing Microengine C Code

- In certain cases, you might want to execute different code in different execution contexts. The context id is obtained by a `ctx()` call. It is preferable to use this call directly in every construct that controls execution flow. Note that when the program is compiled with 4 contexts enabled, the enabled contexts are the even ones (0, 2, 4, 6).
- Unions are a handy way to get around optimization problems with bitfields.
- You can use `structa = structb` as a clean way to copy blocks of registers (xfer to gpr, etc).
- The compiler globally resolves GPRs so that functions can return integral or aggregate values in registers to multiple callers.
- When possible, avoid multiple reads/writes of a struct in memory by first copying it to a temporary struct in local memory or registers, operating upon this struct, and finally writing it out to memory as needed.
- When debugging, always display instruction addresses. If there are no addresses displayed, the compiler may have removed your code.
- Use the `-Qperfinfo` compiler option to analyze register pressure, register spills, and register liveness information. You can also get symbol map and function size data. The data produced by the `-Qperfinfo` option, especially register spill information, is very important. The register allocator assigns variables to a limited number of hardware registers during program execution. When too many registers are in use simultaneously the compiler detects a "register conflict" and instead of assigning a variable to a register demotes it to local memory. In certain

cases the compiler might allocate such a variable in SRAM, which may result in significant performance degradation.

- When performing an asynchronous memory write operation (i.e. a write which waits on "sig_done"), you must call the `__free_write_buffer()` intrinsic to specify the point after which the operation can be considered to have completed. Without this call, the compiler will by default assume that the transfer registers involved in the write operation can be reused immediately after the operation has been issued. This behavior may cause invalid data to be written out to memory. For example:

```
__declspec(sram_write_reg) int x, y;
unsigned int addr = 0x200;
signal s1, s2;

x = 10;
__asm sram[write, x, addr, 0, 1], sig_done[s1];
y = 20;
// x is still alive at this point
__asm sram[write, y, addr, 4, 1], sig_done[s2];
// y is still alive at this point
wait_for_all(&s1, &s2);
// the following calls are need to prevent x, y
// being released before I/O finished.
free_write_buffer(&x); // or __implicit_read(&x)
free_write_buffer(&y); // or __implicit_read(&x)
```

- Asynchronous reads may require the use of `__implicit_read()` or `__free_write_buffer()`, if not all the data is being used. Consider the following example:

```
SIGNAL sig1, sig2;
SIGNAL_PAIR sigpair;
__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, 2, sig_done, &sigpair);
__wait_for_all(&sig1, &sigpair);
sum += sr1[0] + sr2[0]; // only use the first element of each
```

The compiler will see that your program is not using the entirety of the buffers `sr1` and `sr2`. It may attempt to conserve registers by assigning overlapping register ranges to `sr1` and `sr2`. For example, `$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. However, if the `sram_read()` and `dram_read_S()` operations complete out of order, this assignment will cause problems. When the `dram_read_S()` operation completes, the data you need 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`.

You must avoid this situation by informing the compiler that the entirety of both transfer buffers is being used, with the `__free_write_buffer()` or `__implicit_read()` intrinsics:

```
SIGNAL sig1, sig2;
SIGNAL_PAIR sigpair;
__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, 2, sig_done, &sigpair);
__wait_for_all(&sig1, &sigpair);
__implicit_read(sr1); // create a use of the whole four-word buffer
```

```
__implicit_read(sr2); // create a use of the whole four-word buffer
sum += sr1[0] + sr2[0]; // only use the first element of each
```

If the compiler sees that all the data in both `sr1` and `sr2` is being used, it will not attempt to overlap the register assignments for those buffers, and will assign different registers (8 total in this example) to each buffer.

- Intrinsic functions which perform atomic operations (test-and-set, test-and-clear, test-and-add, test-and-sub, hash, put-ring, swap) accept two transfer registers, which are required by the IXP architecture to have the same name (i.e. the read register `$0` must be paired with the write register `$0`). This restriction may create unexpected behavior if the same variable is reused in more than one atomic operation. For example, if `__sram_test_and_set()` is called with arguments `val` and `mask`, and another `__sram_test_and_set()` call is made with arguments `val2` and `mask`, the variables `val` and `val2` will need to be assigned the same register because of their association with the variable `mask`. Different, but equally sized, variables should be used for each atomic operation if this register assignment behavior is not desired. The compiler will print an error message if this situation occurs.
- Passing an address of `xfer/signal/gpr` to a function is generally disallowed unless the callee is an intrinsic function. For `__forceinline` function compiled under command line options other than `-Od` and `-Ob0`, the compiler does its best to propagate `xfer/gpr/signal` with the address taken at the call site to pointer dereferencing inside callee, as if `xfer/gpr/signal` were directly used. If your program saves aside the pointer to another variable, however, the compiler is not always able to remove that statement, which causes an `xfer/gpr/signal` register violation from having its address taken. The general guideline follows:

If a user-defined function `f` takes the address of a `xfer/gpr/signal` variable in parameter `p`, then

- `f` must be inlined, and
- `p` can only be safely used in the following two cases:
 - 1) to de-reference in a form like `"*((optional-cast)p + const-offset)";`
 - 2) to compare `p` against another parameter `q`, which takes address of `xfer/gpr/signal`.

Pointer arithmetic on `p`, if any, can only happen in the above two cases.

Saving aside `p` in another user-defined variable `r` may cause the compiler to crash because statements like `r=p` (or `r=&xfer` after inlining) may not always be removed (especially combining with pointer arithmetic like `r=p+1`, or complex control-flow-graph between that and use of `r`, etc) and violates the rule that `xfer/gpr/signal` cannot have their addresses taken.

- When you use the compiler option `-uc` for mixing C and microcode programming, the compiler assumes C code does not contain the whole program, and makes very conservative optimization decisions. Even if you do have the whole program in C, you might still observe performance loss with `-uc` than without `-uc`. As a result, when you have the whole program in C, always use the compiler to generate a list file directly, and do not use the `-uc` switch.
- Write transfer buffers should be fully initialized, with either an assignment or an `__implicit_write()` call, before they are used in I/O operations. Otherwise, the compiler will assume that the uninitialized transfer registers are actually initialized elsewhere, and will extend the live range of those registers above the declaration point of the buffer. This leads to inefficient register usage and possibly extra spills or a register allocation failure. Example:

```
__declspec(sram_write_reg) int buf[10];
__implicit_write(&buf); // "initialize" the buffer
count = foo();
sram_write(&buf, addr, count, ctx_swap, &sig);
```

```
// I/O size determined at runtime: assumed to be max size
```

Limited forms of indirect register access are possible if you consider the live range computations that the compiler performs, and if you are careful to tell the compiler which registers will be accessed, by using the `__implicit_read()` and `__implicit_write()` intrinsics. For example:

```
__declspec(sram_write_reg) int wbuf[10];
__implicit_write(&wbuf); // Tells compiler that wbuf is being written to
__asm {
... loop which writes "wbuf" using T_INDEX
}
sram_write(&wbuf, addr, 10, ctx_swap, &sig);
```

In the above code, if the `__implicit_write()` call were not present, the compiler would not know that the inline assembly code writes to the buffer `wbuf`. The `sram_write()` call would then appear to be using data defined elsewhere, which may cause incorrect program behavior.

Specifically, the compiler will assume that the values used in the `sram_write()` call are the previous values of the transfer registers allocated to the "wbuf" array, and may propagate those values into the `sram_write()` call. This will overwrite the values written by the `T_INDEX` loop, causing the wrong values to be written to SRAM. `__implicit_read()` is necessary when reading registers with indirect accesses:

```
__declspec(sram_read_reg) int rbuf[10];
sram_read(&rbuf, addr, 10, ctx_swap, &sig);
__asm {... loop which reads "rbuf" using T_INDEX
}
__implicit_read(&rbuf); // tells compiler that "rbuf" is read
```

In the above example, the compiler may remove the `sram_read()` call if it does not see any code which uses the "rbuf" transfer buffer. The `__implicit_read()` call informs it that the buffer is in fact "live" and its contents are needed.

Indirect accesses should not be used to read or write registers assigned to other threads.

- If the compiler cannot determine that the transfer size for a memory I/O or hash operation is a known constant (for example, calling `sram_read()` with a variable "x" for the size argument, where "x" does not have a constant value), the "indirect form" of the underlying I/O instruction will be generated, which allows the size to be determined at runtime. However, the compiler still needs to know how many transfer registers should be reserved for the I/O operation, to prevent values from being accidentally overwritten by other I/O operations. The indirect form of the I/O intrinsics contain a parameter, "max_nn", which allows you to specify this information. When the compiler itself generates the indirect form from a direct I/O call with a non-constant size parameter, the compiler will look at the transfer buffer argument passed in and assume that the entire buffer will be accessed. This assumption may cause problems if your program makes an I/O call with a non-constant size that only writes to part of a buffer, and if your program expects the rest of the buffer to retain its previous value. For example:

```
__declspec(sram_read_reg) a[10];
__sram_read(&a, addr, 10...); // init "a" with 10 words
__sram_read(&a, addr, x....); // read "x" words into "a". Suppose "x" < 5,
//but the compiler does not know it.
... = a[7]
```

In the above example, the second I/O call has a size parameter that the compiler cannot determine is a constant, for whatever reason. Suppose your program knows that this value is never greater than 5. Then you might expect that a [7] will never be touched by the second I/O

operation, and that the value of a [7] will be the one read from the first I/O operation. The compiler, however, cannot perform this analysis, and will assume that all the elements of "a" are written to by the second I/O operation. Therefore, the first `__sram_read()` call will appear to be redundant and may be removed by the compiler, which will cause `a[7]` to have an unknown value.

The compiler cannot detect when the above situation is occurring (otherwise it would not be a problem). It is recommended that you compile your code with the `-Qperinfo=128` option, which generates warnings for all the instances that direct I/O operations are auto-converted into indirect form. You should examine each of the reported instances, determine if they are making the hidden assumption about partial buffer access described above, and, if so, manually change the I/O operation to the "true" indirect form (`sram_read_ind()` in the above example), where the maximum transfer size can be specified as a parameter.

- If inline assembly is used with a comment `“;”` additional C commands after the `“;”` are not seen.

Example: `__asm { pci_dma[d0, d1, order_queue], ctx_swap; }`

This is interpreted by the compiler as:

```
__asm { pci_dma[d0, d1, order_queue], ctx_swap
```

The trailing `“}”` is lost because of the comment start symbol `“;”`

7.3 Troubleshooting

7.3.1 Program Does Not Fit

- Compile for size (`-O1`).
- Use `__noinline` on any functions that are inlined and called from multiple places to prevent the compiler from inlining them.
- Reduce inlining (`-Ob1`, or `-Ob0`).

7.3.2 Program Does Not Run Correctly

- Compile at warning level 4 (`W4`) and check warning messages.
- Generate source level debugging information (`-Zi`).

Note: Refer to [Table 2, “Supported CLI Option Switches” on page 20](#) for more information on compiler command line interface (CLI) option switches.

7.4 Debugging Inline Functions

When a function is inlined, the line number associated with the inlined code is the same as the call site. You can no longer step into a inlined function.

Mutual Exclusion Library

8

8.1 Introduction

The mutual exclusion locks (mutexes) provided in this library are designed to prevent multiple contexts of an ME from simultaneously executing critical sections of code that access shared data. In other words, mutexes are used to serialize the execution of a microengine's contexts.

All mutexes must be global. A successful call for a mutex lock via `MUTEXLV_lock()` will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks it via `mutex_unlock()`. Threads within the same micro engine can share mutexes.

The MUTEXLV uses the microengine C construct:

```
__declspec(shared gp_reg)
```

which is a shared general purpose register (across threads in a ME). If the compiler cannot allocate the `gp_reg` object in a register, it reports an error and aborts.

The MUTEXLV are implemented via macros, as they must manipulate the register object directly. A single MUTEXLV object is capable of 32 mutex(s), each referred to with a user specified unique id (MUTEXID) from [0 .. 31], not necessarily a constant. There is no range checking on the required id.

To coordinate threads on multiple microengines, there are microengine global mutexes.

The MUTEXG uses the microengine C construct:

```
__declspec(import | export)
```

which by default are shared volatile variables across microengines.

MUTEXG_IMPORT and MUTEXG_EXPORT are implemented as macros. They are functionally similar to the MUTEXLV macros, except they are not vector valued (that is, there is only one available per declaration).

Note: There is no corresponding functions for the semaphore library.

8.2 MUTEXLV Usage

```
MUTEXLV lock= 0;
f (MUTEXID handle)
{
    ...
    MUTEXLV_lock(lock, handle)

    // lock data/code access to only one ME local thread
```

```
        MUTEXLV_unlock(lock, handle)
        ...
    }
```

If the handle value is reused, you must use the correct barrier synchronization (semaphore).

8.3 MUTEXG Usage

```
MUTEXG lock= 0;
f(MUTEXID handle) {
    ...
    MUTEXG_lock(lock, handle)

    // lock data/code access to one ME thread
    MUTEXG_unlock(lock, handle)
    ...
}
```

8.4 Functions

8.4.1 MUTEXLV_init (MUTEXLV)

PARAMETERS

MUTEXLV: mutex object to be initialized

DESCRIPTION

Initialize all ids to zero.

This is very hard to use, as you must insure the mutex is initialized only once (either globally, or with some semaphore)

ERRCODE

No error code is returned

8.4.2 MUTEXLV_destroy(MUTEXLV,MUTEXID)

PARAMETERS

MUTEXLV: mutex object

MUTEXID: handle to specific mutex id

DESCRIPTION

This clears the specific [mutex, id] in mutex.

ERRCODE

No error code is returned

8.4.3 MUTEXLV_lock(MUTEXLV, MUTEXID)

PARAMETERS

MUTEXLV: mutex object
MUTEXID: handle to specific mutex id

DESCRIPTION

This tests and blocks a specific [mutex, id] for a lock. If busy, the current context is swapped out. If free, the [mutex,id] is set and `ERRCODE_EOK` is returned in supplied argument. There is no sense of `ctx()` ownership or recursion. If the same thread tries to reacquire the lock (that it already owns), it too will block.

ERRCODE

No error code is returned

8.4.4 MUTEXLV_unlock(MUTEXLV, MUTEXID)

PARAMETERS

MUTEXLV: mutex object
MUTEXID: handle to specific mutex id

DESCRIPTION

Unlock the [mutex, id]. There is no sense of `ctx()` ownership (i.e. a different thread may unlock the object)

ERRCODE

No error code is returned

8.4.5 MUTEXLV_trylock(MUTEXLV, MUTEXID, ERRCODE)

PARAMETERS

MUTEXLV: mutex object
MUTEXID: handle to specific mutex id
ERRCODE: specific error code returned

DESCRIPTION

This tries to acquire the [mutex, id] (if free) but does not block.

ERRCODE

`ERRCODE_EBUSY`: lock is busy
`ERRCODE_EOK`: lock has been acquired

8.4.6 MUTEXLV_testlock(MUTEXLV, MUTEXID, ERRCODE)

PARAMETERS

MUTEXLV: mutex object
MUTEXID: handle to specific mutex id
ERRCODE: specific error code returned

DESCRIPTION

This tests but does not acquire the [mutex, id]

ERRCODE

ERRCODE_EBUSY: lock is currently spinning

ERRCODE_EOK: lock is free (i.e. may be acquired)

8.4.7 MUTEXG_init (MUTEXG)

PARAMETERS

MUTEXG: mutex object to be initialized.

DESCRIPTION

Initialize to zero. You must insure the mutex is initialized only once, either globally or through a semaphore.

ERRCODE

No error code is returned.

8.4.8 MUTEXG_destroy (MUTEXG)

PARAMETERS

MUTEXG: mutex object.

DESCRIPTION

Clears the specified mutex object.

ERRCODE

No error code is returned.

8.4.9 MUTEXG_lock (MUTEXG)

PARAMETERS

MUTEXG: mutex object.

DESCRIPTION

This tests and blocks a specific mutex for a lock. If busy, the current context is swapped out. If free, the mutex is set and ERRCODE_EOK is returned in the supplied argument. There is no concept of ctx() ownership or recursion. If the same thread tries to re-acquire the lock that it already owns, it too will block.

ERRCODE

No error code is returned.

8.4.10 MUTEXG_unlock (MUTEXG)

PARAMETERS

MUTEXG: mutex object.

DESCRIPTION

Unlock the mutex. There is no concept of ctx() ownership (that is, a different thread may unlock the object).

ERRCODE

No error code is returned.

8.4.11 MUTEXG_trylock (MUTEXG, ERRCODE)

PARAMETERS

MUTEXG: mutex object.

ERRCODE: specific error code returned.

DESCRIPTION

This attempts to acquire the [mutex, id] if it is free, but does not block.

ERRCODE

ERRCODE_EBUSY: Lock is busy.

ERRCODE_EOK The lock has been acquired.

8.4.12 MUTEXG_testlock (MUTEXG, ERRCODE)

PARAMETERS

MUTEXG: mutex object.

ERRCODE: specific error code returned.

DESCRIPTION

This tests but does not acquire the [mutex, id].

ERRCODE

ERRCODE_EBUSY: Lock is currently spinning.

ERRCODE_EOK The lock is free (that is, it may be acquired).



Semaphore Library

9

9.1 Semaphore Data Types

```
typedef unsigned int SEMVALUE;
```

```
typedef volatile __declspec(shared gp_reg) struct SEML
{
    unsigned barrier:1;
    unsigned init:1;
    unsigned reserved:14;
    unsigned initval:8;
    unsigned val:8;
} SEML;
```

9.2 Semaphore Functions

9.2.1 SEML_init(SEML, SEMVALUE)

PARAMETERS

SEML: semaphore object
SEMVALUE: initial value of the semaphore counter (max 0xff)

DESCRIPTION

This function initializes an unnamed semaphore. The initial value of the semaphore is set to 'value'. The semaphore may also be initialized globally by

```
SEML sem= SEML_init_list(value);
```

ERRCODE

No error code is returned.

9.2.2 SEML_destroy(SEML)

PARAMETERS

SEML: semaphore object

DESCRIPTION

This function destroys an unnamed semaphore.

ERRCODE

No error code is returned.

9.2.3 SEML_post(SEML) SEML_dec(SEML)

PARAMETERS

SEML: semaphore object

DESCRIPTION

This function will post (or dec) a wakeup to a semaphore. If there are waiting threads, one is unblocked; otherwise, the semaphore value is incremented (decremented) by one.

ERRCODE

No error code is returned.

9.2.4 SEML_wait(SEML)

PARAMETERS

SEML: semaphore object

DESCRIPTION

This function waits on a semaphore. If the semaphore value is greater than zero, it decreases its value by one. If the semaphore value is less than or equal zero, then the calling thread is blocked until it can successfully decrease the value.

ERRCODE

No error code returned.

9.2.5 SEML_trywait(SEML, ERRCODE)

PARAMETERS

SEML: semaphore object

ERRCODE:

DESCRIPTION

Similar to SEML_wait except that if the semaphore value is zero, then this function returns immediately with the error EAGAIN.

ERRCODE

ERRCODE_EAGAIN: the semaphore was already locked,

ERRCODE_OK:

9.2.6 SEML_barrier(SEML,n)

PARAMETERS

SEML: semaphore object

DESCRIPTION

This function performs the dual of SEML_wait. It is used to provide a synchronization point for threads to join by waiting on a semaphore. If the semaphore value is greater than zero, then the calling thread is blocked until it can successfully increase the value. If the semaphore value is zero, it increases its value by one.

ERRCODE

No error code returned.

9.2.7 SEML_trybarrier(SEML, ERRCODE)

PARAMETERS

SEML: semaphore object

ERRCODE:

DESCRIPTION

Similar to SEML_barrier except that if the semaphore value is less than or equal zero, then this function increments the value.

ERRCODE

ERRCODE_EAGAIN:the semaphore was already locked,

ERRCODE_OK:

9.2.8 SEML_getvalue(SEML)

PARAMETERS

SEML: semaphore object

DESCRIPTION

Return the value associated with the semaphore

ERRCODE

No error code returned.

9.2.9 SEML_set_barrier_at(SEML,n) SEML_clr_barrier_at(SEML,n)

Auxiliary macros used by semaphore macros.



-Qperinfo Output Information

A

This appendix provides additional information for the -Qperinfo command line option switch.

A.1 -Qperinfo=1

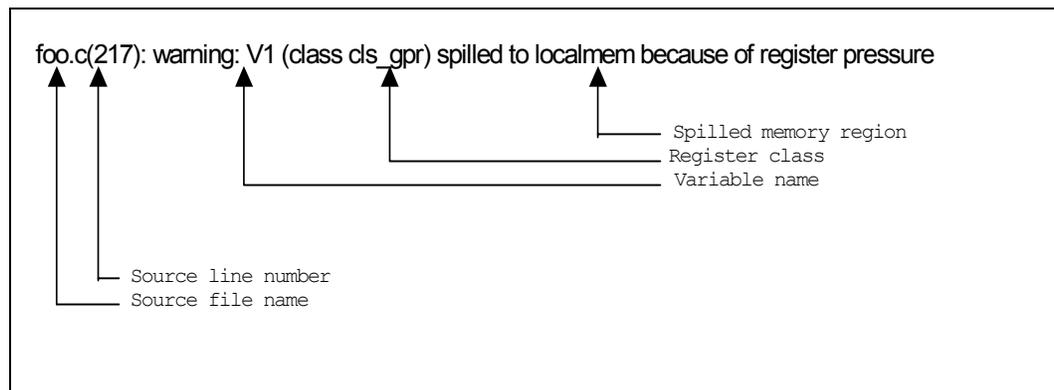
Function: Variable spill

Description:

Provides information for all GPR variables spilled into local memory or SRAM during register allocation.

Note: This does not include variables spilled into NN (next neighbor) registers.

Format:



Example:

```
foo.c(217): warning: V1 (class cls_gpr) spilled to localmem because of register pressure.
foo.c(218): warning: V2 (class cls_gpr) spilled to localmem because of register pressure.
foo.c(219): warning: V3 (class cls_gpr) spilled to localmem because of register pressure.
foo.c(220): warning: V4 (class cls_gpr) spilled to sram because of register pressure.
foo.c(221): warning: V5 (class cls_gpr) spilled to sram because of register pressure.
foo.c(222): warning: V6 (class cls_gpr) spilled to sram because of register pressure.
```

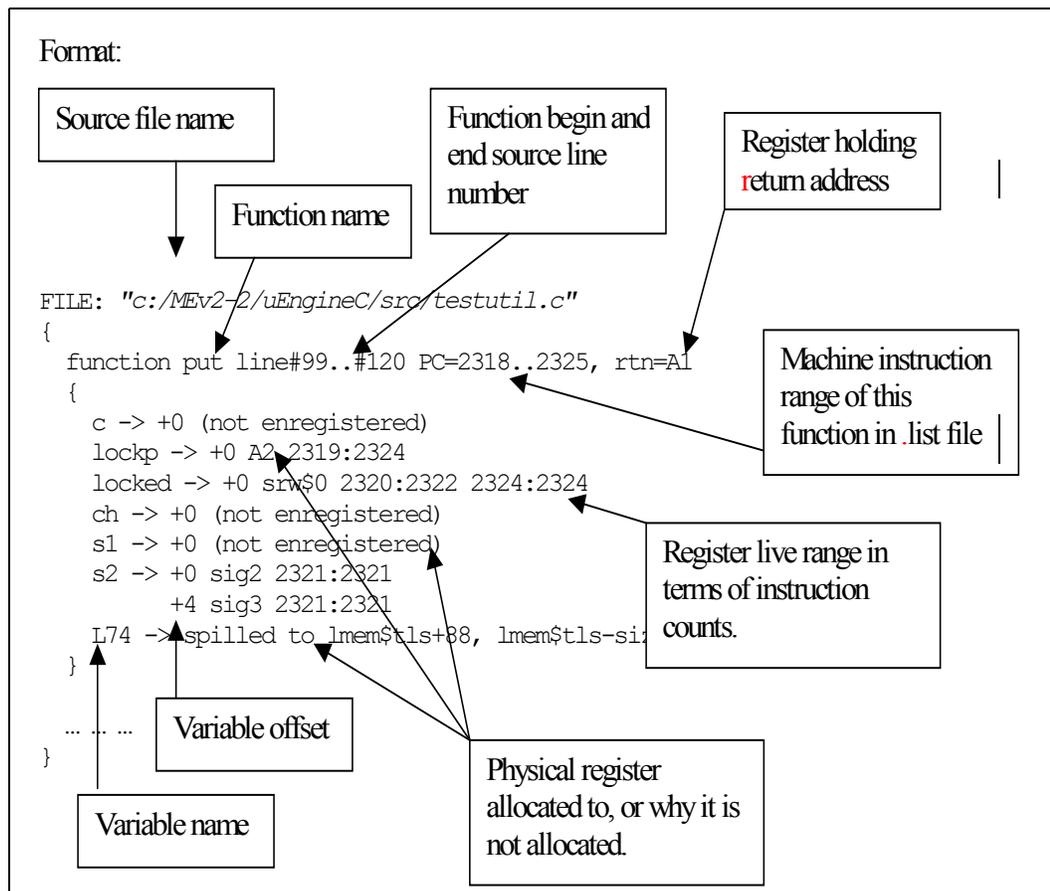
A.2 -Qperfinfo=2

Function: Live range and allocation

Description:

Provides instruction-level symbol liveliness and register allocation.

Format:



Notes:

- Live range is in the format of `beginning_instruction_#:end_instruction_#`.
- Some variables are not allocated to registers because compiler removed this variable, or it is merged into another variable.
- Some variables do not have a recognizable name. They are compiler generated.
- Physical registers format:
An: A bank register

Bn: B bank register
 sign: signal register
 sr\$n: sram read register
 dr\$n: dram read register
 sw\$n: sram write register
 dw\$n: dram write register
 srw#n: sram read/write register

Example:

```
//-----
{
FILE: "largeRegSpill_0_1.c"
{
function main line#28..#1006 PC=0..2315, rtn=A0
{
L0 -> spilled to lmem$tls+0, lmem$tls-size=256
NOTE: lmem$tls start = 0
L1 -> spilled to lmem$tls+4, lmem$tls-size=256
L2 -> spilled to lmem$tls+8, lmem$tls-size=256
L3 -> spilled to lmem$tls+12, lmem$tls-size=256
L4 -> spilled to lmem$tls+16, lmem$tls-size=256
L84 -> +0 A24 157:1543
L85 -> +0 B23 160:1552
L86 -> +0 A23 163:1561
L128 -> +0 A1 244:1939
L129 -> +0 A0 245:1948
L130 -> spilled to lmem$tls+92, lmem$tls-size=256
L131 -> spilled to lmem$tls+96, lmem$tls-size=256
L133 -> spilled to lmem$tls+104, lmem$tls-size=256
}
}
}
FILE: "c:\MEv2-2\uEngineC\src\rtl.c"
{
function exit line#50..#65 PC=2316..2317, rtn=A0
{
status -> +0 (not enregistered)
}
}
FILE: "c:\MEv2-2\uEngineC\src\testutil.c"
{
function put line#99..#120 PC=2318..2325, rtn=A1
{
c -> +0 (not enregistered)
lockp -> +0 A2 2319:2324
locked -> +0 srw$0 2320:2322 2324:2324
ch -> +0 (not enregistered)
s1 -> +0 (not enregistered)
s2 -> +0 sig2 2321:2321
+4 sig3 2321:2321
}
}
function putui line#179..#199 PC=2326..2335, rtn=A1
{
val -> +0 (not enregistered)
lockp -> +0 A2 2327:2334
locked -> +0 srw$0 2329:2331 2334:2334
h -> +0 sw$1 2328:2332
s1 -> +0 (not enregistered)
s2 -> +0 sig2 2330:2330
+4 sig3 2330:2330
}
}
}
```



A.3 -Qperfinfo=4

Function: (None)

Description:

This option has been deprecated and replaced with the -Qliveinfo option. Please see [Section 3.9](#), “[Viewing Live Ranges](#)” on [page 57](#) for more information.:

A.4 -Qperfinfo=8

Function: Function size

Description:

Provides the number of instructions in each function.

Example:

function:	size
_exit:	2
_put:	8
_putui:	10
_mput\$5:	16
_putns:	38
_putsi:	13
_main:	2316
Total size:	2403

A.5 -Qperfinfo=16

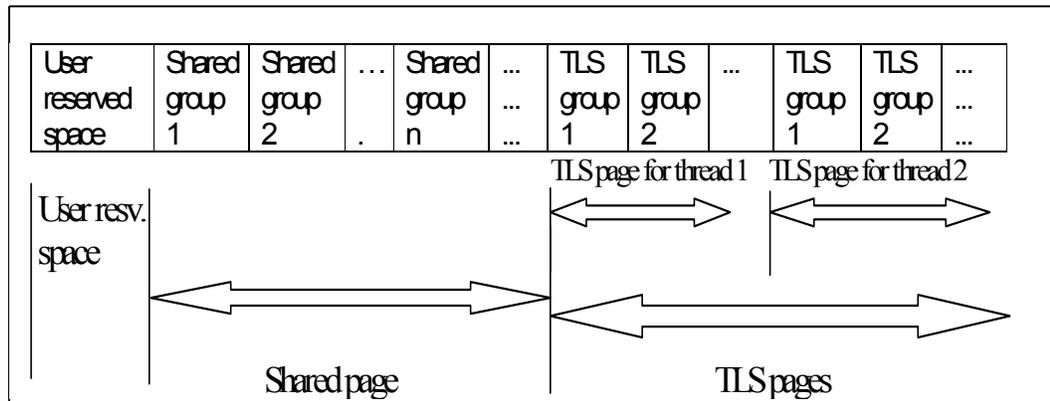
Function: Local memory allocation

Description:

Provides information about how the compiler allocates local memory. Each local memory variable (either user-defined or a compiler-generated spill variable) belongs to a local memory group. Each local memory group contains one or several local memory variables allocated within a 64-byte range. Access to two local variables in the same group can share the same local memory base pointer if its value is still valid.

Eventually, all the groups are divided into two categories: thread-local group and shared group. All threads share the shared groups, and each thread has its own copy of the thread-local groups.

Layout (TLS for thread-local-storage):



Notes:

- You can reserve an area of local memory from compiler allocation through a command line option.
- The offset field in the printout is the offset from the beginning of the thread-local page for thread-local data, or from the beginning of the shared page for shared data. They are not offsets within the group.

Example:

=> User reserved: 0 bytes, Shared segment: 64 bytes, Local page (including gap): 320 bytes

=> Gap between context pages is 40 bytes The data on the page is 280 bytes

```
Direct access local mem group 0x14ea3f8
Maximum offset used: 60 Alignment: 64
Num members: 1 Total size: 128
[This group contains thread local symbols]
```

copylmemsram.c(126): tmp allocated at offset 0

Direct access local mem group 0x1516c04
Maximum offset used : 12 Alignment: 4
Num members: 4 Total size: 16
[This group contains thread local symbols]

copylmemsram.c(240): _m1 allocated at offset 128
copylmemsram.c(241): _m2 allocated at offset 132
copylmemsram.c(242): _m3 allocated at offset 136
copylmemsram.c(243): _m4 allocated at offset 140

Direct access local mem group 0x1516dfc
Maximum offset used: 12 Alignment: 4
Num members: 4 Total size: 16
[This group contains shared symbols]

copylmemsram.c(244): _p1 allocated at offset 0
copylmemsram.c(244): _p2 allocated at offset 4
copylmemsram.c(244): _p3 allocated at offset 8
copylmemsram.c(244): _p4 allocated at offset 12

A.6 -Qperfinfo=32

Function: Interference information for variables spilled into SRAM.

Description:

When a GPR variable is spilled into SRAM, this information prints out all the other variables that interfere with this one. Two variables interfere with each other when they can not be allocated to the same register.

Example:

```
foo.c(221):L24 conflicts with:  
foo.c(197):L0 foo.c(198):L1 foo.c(199):L2 foo.c(200):L3  
foo.c(201):L4 foo.c(202):L5
```

Note: Variables are in the format: <source file>(<source line>):<name>

A.7 -Qperfinfo=64

Function: Scheduler statistics

Description:

The instruction scheduler moves instructions to fill delay slots. The -Qperfinfo printout for the scheduler looks like the following example.

Example:

```
/*  
 * Scheduler Summary  
 */
```

Nop(s) removed: 53 (8.5% of total 624)

In this example, there were a total 624 nops in the whole program, and the compiler was able to remove 53 of them.

A.8 -Qperfinfo=128

Function: Warn if the compiler cannot detect the I/O buffer size

Description:

The I/O intrinsic functions such as `sram_write()` allow you to pass a non-constant count argument. In this case, the compiler will generate the “indirect” form of the I/O instruction (see the Programmer's Reference Manual for details). However, the compiler will need to know how many transfer registers to reserve for the I/O transfer. Since the compiler does not know the exact reference count, it will assume that the whole variable is used in the intrinsic.

Example:

```
int __declspec(sram_write_reg) rr[8];  
..  
sram_write(&rr, addr, size, ctx_swap, &s1);  
// size is unknown at compile time
```

The compiler will reserve 8 transfer registers for the SRAM transfer in this example, although at runtime only a smaller number may be needed. `-Qperfinfo=128` will print warnings when the compiler detects a situation similar to the above.

Example:

```
C:\CVS\include\align.h(124): 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.
```

A.9 -Qperfinfo=256

Function: Display information on restrict pointer optimization

Description:

The compiler can optimize dereferences of pointer parameters declared with the “restrict” keyword (see [Section 3.7.4.1, “The “restrict” Qualifier” on page 54](#) for details). This allows objects passed to such functions to be allocated to registers. Not all "restrict" pointer parameters can be optimized. The -Qperfinfo=256 option shows which parameters are optimized, and provides information about the parameters which are not optimized.

The argument name may be printed in its internal modified form. It has the format:
<user_name>_<compiler_mangling_string>.

Example:

Pointer argument xyz_379_V\$200\$1\$1 in function _foo was optimized and dereferences were eliminated.

A.10 -Qperfinfo=512

Function: Compiler printout for jump target offsets.

Description:

The compiler supports the `jump[]` instruction in inline assembly code. You have to make sure that the offsets passed to the `jump[]` instruction match valid target labels in the inline assembly. -Qperfinfo=512 provides the offset of each symbolic label listed in a jump target list.

Example:

For code like:

```
__asm __attribute(ASM_HAS_JUMP)
{
    br[jmp]
base0:
    immed[result, 1]
    br[last]
base1:
    immed[result, 2]
    br[last]
    jmp:
    jump[offset, base0], targets [base0, base1, base2, base3]
base2:
    immed[result, 3]
    call[foo2]
    br[last]
base3:
    immed[result, 4]
    call[foo2]
    br[last]
last:
}
```

Compiler -Qperfinfo=512 printout has the following appearance:

```
test_jump.c(55): inline-asm jump may have potential offset(s) = 0, 2, 5, 9
```

A.11 -Qperfinfo=1024

Function: Boolean propagation optimization

Description:

The compiler performs an optimization that determines the value of a constant conditional based on the result of other conditionals.

Example:

```
if (1) {  
    x = 5;  
}  
else {  
    x = 6;  
}  
if (x == 5) {  
    ...  
}
```

When this optimization occurs, this `_Qperfinfo` option displays results as the following example shows:

```
[1] Bool_prop transformation performed  
[2] Bool_prop transformation performed
```

A.12 -Qperfinfo=2048

Function: Register requirements report

Description:

This -Qperfinfo option displays a report on areas of your program that require a large number of “live” registers, indicating possible spill areas. For a more detailed description of the concepts of liveness and spilling, please refer to [Section 3.8](#).

Example:

```
----- GPR Requirements Report -----  
  
0 relative A bank GPRs were used for shared variables  
0 relative B bank GPRs were used for shared variables  
3 thread local variables can be colored using abs registers  
1 relative A bank GPRs used as absolute regs  
1 relative B bank GPRs used as absolute regs  
15 relative A registers available for coloring  
15 relative B registers available for coloring  
409 variables colored with relative GPRs  
59 A bank and 57 B bank GPRs were needed for coloring  
118 total GPRs needed to color without any spilling  
  
Start of high GPR usage region in function _mput$5  
Starting at c:\\clin1\\ixp1200\\uEngineC\\src\\../samples/util/util.c line 414  
    End of high GPR usage region at  
c:\\clin1\\ixp1200\\uEngineC\\src\\../samples/util/util.c 415  
Max usage is (115) at c:\\clin1\\ixp1200\\uEngineC\\src\\../samples/util/util.c  
415
```

(Note that formatting can vary, depending upon text content and line breaks.)

“Coloring” is the process of assigning registers to variables. In the above example, all program variables must be assigned to one of 30 registers (15 in each bank). The region of the program between lines 414 and 415 in the file util.c would require 115 free registers to avoid spilling. Since the number of registers is fixed by the architecture, if you wish to avoid spilling, the program code or data must be restructured so that $115 - 30 = 85$ fewer words of data are live in the indicated region.

A.13 -Qperfinfo=4096

Function: Switch optimization report

Description:

The compiler can perform an optimization, described in [Chapter 3, “User-Guided switch\(\) Statement Optimization”](#), which creates faster code for switch() statements. This -Qperfinfo option generates a report on which switch() statements in your program were optimized, and how the jump[] calculation was performed.

Example:

```
/*
 * Switch Pack Report
 */

Function: _main,
        .switch[$0, SW, l_11#, swpack_auto, nlive=1, a1, b0, a0]
Switch has 3 target(s); Min/Max-distance = 2/2
Estimated PC(s): 31 33 35
0 nop(s) needed
Sequence to compute new jump index y(b0) = x($0) * 2: (t = a1)
        y = x<<1
>>> Pack it
```

A.14 -Qperfinfo=8192

Function: Print parallelization summary information

Description:

When I/O parallelization is enabled with the -Qnew_opt switch, this -Qperfinfo option prints summary information on that optimization.

Example:

```
----- I/O parallelization report -----  
Total number of 2 I/O instructions are parallelized  
    File foo.c, at line 27  
    File foo.c, at line 31  
-----
```

A

- alignment of data types 35
- allocation attributes 40
- allocation region 40
- arrays of transfer registers 42
- assembly language
 - inline 357
 - restrictions 365

B

- bitfields 33
- block of __asm assembly code 359

C

CAP data types 247

- cap_csr_t 247
- cap_read_write_ind_t 249
- local_csr_t 248

CLI option switches

- ? 20
- c 20
- Dname[=value] 20
- DSDK_3_0_COMPATIBLE 21
- E 21
- EP 21
- Fa<filename> 21
- Fe<file> 21
- FI<file> 21
- Fi<file> 21
- Fo<Dir\> 21
- Fo<file> 21
- Gx2400 21
- help 20
- I path[;path2...] 21
- link[linker options] 21
- Obn 21
- On 21
- P 21
- Qbigendian 21
- Qdefault_sr_channel=<0...3> 21
- Qerrata 21
- Qip_no_inlining 22
- Qlittleendian 22
- Qliveinfo 22
- Qliveinfo=gr,sr,... 22

- Qlm_start=<n> 22
- Qlm_unsafe_addr 22
- Qlmpt_reserve 22
- Qmapvr 22
- Qnctx=<1, 2, 3, 4, 5, 6, 7, 8> 22
- Qnctx_mode=<4, 8> 22
- Qnn_mode=<0, 1> 22
- Qnolur=<func_name> 22
- Qold_revision_scheme 22
- Qperfinfo=n 23
- Qrevision_max=m 23
- Qrevision_min=n 23
- Qspill=<n> 23
- s 23
- uc 24
- Wn n=0, 1, 2, 3, 4 24
- Zi 24
- coloring 410
- command line 19
 - command line options
 - Qperfinfo=8192 412
- command line options
 - Qperfinfo=1 397
 - Qperfinfo=1028 409
 - Qperfinfo=128 406
 - Qperfinfo=16 402
 - Qperfinfo=2 398
 - Qperfinfo=2048 410
 - Qperfinfo=256 407
 - Qperfinfo=32 404
 - Qperfinfo=4096 411
 - Qperfinfo=512 408
 - Qperfinfo=64 405
 - Qperfinfo=8 401
- compilation model 18
- compiler optimizations 367
- context relative variables 51
- context swap 44
- control and status register (CSR) access functions 247
- conventions 12
- CRC functions
 - crc_read() 308
 - crc_write() 309

critical path annotation and code layout 59

D

data allocation 40

data terminology 13

 byte 13

 longword 13

 quadword 13

 word 13

data type alignment 35

data type size 34

data types

 basic

 char 33

 enum 33

 int 33

 long 33

 long long 33

 pointers 33

 short 33

debugging 30

debugging inline functions 378

default case removal 63

DRAM

 partial writes 39

DRAM operations 204

 dram_rbuf_read_ind() 209

 dram_read() 205

 dram_read_ind() 206

 dram_read_S() 205

 dram_read_S_ind() 206

 dram_tbuf_write_ind() 210

 dram_write_ind() 208

 dram_write_S() 207

 dram_write_S_ind() 208

E

endian support 38

environment variables 24

exported variables 51

expressions 52

extended function attributes 53

external memory 16

F

FIFO 16

- file types
 - input and output 25
- floating point 34
- function parameter passing 69
- functions 52
- G
- general purpose registers 41
- global data 45
- H
- hash instructions 39
- I
- indirect register access 17
- inline immediate operands 361, 362
- inline instruction format 360
- inline operand syntax 361
- inline register operands 361
- inline usage examples 362
- input and output file types 25
- intrinsic function arguments 354
- intrinsic functions 73
- L
- limitations and restrictions on viewing live ranges 58
- limitations on some I/O functions 232
- live range analysis 55
- liveness computation 56
- load time constants 45
- local memory 16
- local memory allocation 47
- local memory usage 48
- loop unrolling control 64
- M
- machine independent optimizations 367
- memory
 - external 16
 - local 16
- memory I/O data types 83
 - bytes_specifier_t 86
 - reflect_signal_t 87
 - sync_t 85
- memory I/O functions 82, 99
 - scratch operations 99
 - scratch_add(112
 - scratch_add_D() 112

scratch_add_D_ind() 113
 scratch_add_ind() 113
 scratch_clear_bits() 118
 scratch_clear_bits_D() 118
 scratch_clear_bits_D_ind() 119
 scratch_clear_bits_ind() 119
 scratch_decr() 110
 scratch_decr_ind() 111
 scratch_get_ring() 134
 scratch_get_ring_D() 134
 scratch_get_ring_D_ind() 135
 scratch_get_ring_ind() 135
 scratch_incr() 108
 scratch_incr_ind() 109
 scratch_put_ring() 136
 scratch_put_ring_D() 136
 scratch_put_ring_D_ind() 137
 scratch_put_ring_ind() 137
 scratch_read 104
 scratch_read_D 104
 scratch_read_D_ind() 105
 scratch_read_ind() 105
 scratch_set_bits() 116
 scratch_set_bits_D() 116
 scratch_set_bits_D_ind() 117
 scratch_set_bits_ind() 117
 scratch_sub() 114
 scratch_sub_D() 114
 scratch_sub_D_ind() 115
 scratch_sub_ind() 115
 scratch_swap() 132
 scratch_swap_D() 132
 scratch_swap_D_ind() 133
 scratch_swap_ind() 133
 scratch_test_and_add() 124
 scratch_test_and_add_D() 124
 scratch_test_and_add_D_ind() 125
 scratch_test_and_add_ind() 125
 scratch_test_and_clear_bits() 122
 scratch_test_and_clear_bits_D() 122
 scratch_test_and_clear_bits_D_ind() 123
 scratch_test_and_clear_bits_ind() 123
 scratch_test_and_decr() 130

- scratch_test_and_decr_D() 130
- scratch_test_and_decr_D_ind() 131
- scratch_test_and_decr_ind() 131
- scratch_test_and_incr() 128
- scratch_test_and_incr_D 128
- scratch_test_and_incr_D_ind() 129
- scratch_test_and_incr_ind() 129
- scratch_test_and_set_bits() 120
- scratch_test_and_set_bits_D() 120
- scratch_test_and_set_bits_D_ind() 121
- scratch_test_and_set_bits_ind() 121
- scratch_test_and_sub() 126
- scratch_test_and_sub_D() 126
- scratch_test_and_sub_D_ind() 127
- scratch_test_and_sub_ind() 127
- scratch_write() 106
- scratch_write_D() 106
- scratch_write_D_ind() 107
- scratch_write_ind() 107
- memory IO data types
 - cap_csr_read_write_ind_t 93
 - cap_read_write_ind_t 89
 - dram_rbuf_tbuf_ind_t 94
 - dram_read_write_ind_t 91
 - generic_ind_t 98
 - hash_ind_t 97
 - msf_read_write_ind_t 89
 - pci_read_write_ind_t 88
 - reflect_read_write_ind_t 89
 - scratch_atomic_ind_t 93
 - scratch_ring_ind_t 89
 - sram_atomic_ind_t 92
 - sram_csr_read_write_ind_t 93
 - sram_dequeue_ind_t 96
 - sram_enqueue_ind_t 95
 - sram_journal_ind_t 89
 - sram_read_qdesc_ind_t 89
 - sram_read_write_ind_t 88
 - sram_ring_ind_t 89
- memory IO functions 99
- memory regions 43
- memory type modifier 44
- Microengine C compiler

- features 11
- nonfeatures 11
- miscellaneous functions 310
 - __assign_relative_register() 334
 - __critical_path() 348
 - __ctx() 315
 - __free_write_buffer() 337
 - __global_label() 322
 - __implicit_read() 335
 - __implicit_write() 336
 - __impossible_path 351
 - __LoadTimeConstant() 321
 - __ME() 316
 - __n_ctx() 317
 - __nctx_mode() 318
 - __no_spill_begin() 345
 - __no_spill_end() 346
 - __no_swap_begin 352
 - __no_swap_end 353
 - __profile_count_start() 331
 - __profile_count_stop 331
 - __set_profile_count() 330
 - __set_timestamp() 328
 - __signal_number() 332
 - __switch_pack 350
 - __timestamp_start() 329
 - __timestamp_stop 329
 - __xfer_reg_number() 333
- assert() 347
- bit_test() 339
- byte_align_block_be() 344
- byte_align_block_le() 343
- dbl_shl() 314
- dbl_shr() 313
- ffs() 320
- inp_state_test() 338
- multiply_16x16() 324
- multiply_24x8() 323
- multiply_32x32() 327
- multiply_32x32_hi() 326
- multiply_32x32_lo() 325
- n_ring_enqueue_incr() 342
- nn_ring_dequeue() 341

- nn_ring_dequeue_incr() 340
- pop_count() 349
- sleep() 319
- Mixed 69
- mixed C and microcode examples 69
- mixing C/microcode in one microengine 66
- MSF operations 211
 - msf_fast_write() 221
 - msf_read_D_ind() 214
 - msf_read_ind() 214
 - msf_read64() 215
 - msf_read64_D() 215
 - msf_read64_ind() 216
 - msf_write() 217
 - msf_write_D() 217
 - msf_write_D_ind() 218
 - msf_write_ind() 218
 - msf_write64() 219
 - msf_write64_D() 219
 - msf_write64_D_ind() 220
 - msf_write64_ind() 220
 - sf_read64_D_ind() 216
- multiple critical paths 61
- MUTEX functions
 - MUTEXG_destroy (MUTEXG) 382
 - MUTEXG_init (MUTEXG) 382
 - MUTEXG_lock (MUTEXG) 382
 - MUTEXG_testlock (MUTEXG, ERRCODE) 383
 - MUTEXG_trylock (MUTEXG, ERRCODE) 383
 - MUTEXG_unlock (MUTEXG) 383
 - MUTEXLV_destroy(MUTEXLV,MUTEXID) 380
 - MUTEXLV_init (MUTEXLV) 380
 - MUTEXLV_lock(MUTEXLV) 381
 - MUTEXLV_testlock(MUTEXLV, MUTEXID, ERRCODE) 381
 - MUTEXLV_trylock(MUTEXLV, MUTEXID, ERRCODE) 381
 - MUTEXLV_unlock(MUTEXLV, MUTEXID) 381
- MUTEXG usage 380
- MUTEXLV usage 379
- mutual exclusion library 379
- N
- neighbor mode 42
- network processor specific optimizations 367
- Next Neighbor registers 14, 42

NPU optimizations

- defer slot filling 368
- I/O parallelization 370
- local memory autoincrement/autodecrement conversion 369
- local memory grouping 368
- peephole optimization 368
- read/write combining 367
- registrations 367
- scheduling 370

O

optimizations

- default case removal 63
 - machine independent 367
 - switch block packing 63
- optimizing code 62, 373
- optimizing pointer arguments 53

P

packed aggregates 36

PCI operations 222

- pci_read() 223
- pci_read_D() 223
- pci_read_D_ind() 224
- pci_read_ind() 224
- pci_write() 225
- pci_write_D() 225
- pci_write_D_ind() 226
- pci_write_ind() 226

placement of variables 47

pointer representation 33

program does not fit 378

Q

qualifiers

- restrict 54

R

read transfer register 41

Reflector 17

- inputs/outputs 51

reflector operation 51

Reflector operations

- (summary table) 227
- reflect_read() 228
- reflect_read_D() 228
- reflect_read_D_ind() 229

- reflect_read_ind() 229
- reflect_write() 230
- reflect_write_D() 230
- reflect_write_D_ind() 231
- reflect_write_ind() 231
- register model 13
- register regions 41
- register usage 70
- register variable naming conventions 67
- registers
 - Next Neighbor 14
 - transfer 41
 - volatile 43
- remote read transfer register 51
- restrict qualifier 54
- restrictions on intrinsics 354
- restrictions on mixed C/microcode 70
- running 30
- S
- self mode 42
- semaphore data types 385
- semaphore functions
 - SEML_barrier(SEML,n) 387
 - SEML_destroy(SEML) 385
 - SEML_getvalue(SEML) 387
 - SEML_init(SEML) 385
 - SEML_init(SEML, SEMVALUE) 385
 - SEML_post(SEML) SEML_dec(SEML) 386
 - SEML_set_barrier_at(SEML,n) SEML_clr_barrier_at(SEML,n) 387
 - SEML_trybarrier(SEML, ERRCODE) 387
 - SEML_trywait(SEML, ERRCODE) 386
 - SEML_wait(SEML) 386
- shared data 44
- shared storage 48
- signal variable restrictions 46
- signal variables 45, 46
- signals 17, 45
- single __asm instruction 358
- SRAM operations 138
 - sram_add() 153
 - sram_add_D() 153
 - sram_add_D_ind() 154
 - sram_add_ind() 154

sram_add_int() 197
sram_clear_bit_pos() 198
sram_clear_bits() 151
sram_clear_bits_D() 151
sram_clear_bits_D_ind() 152
sram_clear_bits_ind() 152
sram_csr_read() 171
sram_csr_read_D() 171
sram_csr_read_D_ind() 172
sram_csr_read_ind() 172
sram_csr_write() 173
sram_csr_write_D() 173
sram_csr_write_D_ind() 174
sram_csr_write_ind() 174
sram_decr() 157
sram_decr_ind() 158
sram_dequeue(), sram_dequeue_D() 187
sram_dequeue_D_ind() 188
sram_dequeue_ind() 188
sram_enqueue() 183
sram_enqueue_ind() 184
sram_enqueue_tail() 185
sram_enqueue_tail_ind() 186
sram_fast_journal() 195
sram_fast_journal_ind() 196
sram_get_ring() 189
sram_get_ring_D() 189
sram_get_ring_D_ind() 190
sram_get_ring_ind() 190
sram_incr() 155
sram_incr_ind() 156
sram_journal_D() 193
sram_journal_D_ind() 194
sram_journal_ind() 194
sram_journav() 193
sram_put_ring() 191
sram_put_ring_D() 191
sram_put_ring_D_ind() 192
sram_put_ring_ind() 192
sram_read() 145
sram_read_D() 145
sram_read_D_ind() 146
sram_read_ind() 146

sram_read_qdesc_head(175
sram_read_qdesc_head_D() 175
sram_read_qdesc_head_D_ind() 176
sram_read_qdesc_head_ind() 176
sram_read_qdesc_other() 179
sram_read_qdesc_other_ind(180
sram_read_qdesc_tail() 177
sram_read_qdesc_tail_D() 177
sram_read_qdesc_tail_D_ind() 178
sram_read_qdesc_tail_ind() 178
sram_set_bit_pos() 199
sram_set_bits() 149
sram_set_bits_D() 149
sram_set_bits_D_ind() 150
sram_set_bits_ind() 150
sram_swap() 159
sram_swap_D() 159
sram_swap_D_ind() 160
sram_swap_ind() 160
sram_swap_int() 200
sram_swap_int_D(200
sram_test_and_add() 165
sram_test_and_add_D() 165
sram_test_and_add_D_ind(166
sram_test_and_add_ind() 166
sram_test_and_add_int() 201
sram_test_and_add_int_D() 201
sram_test_and_clear_bit_pos() 202
sram_test_and_clear_bit_pos_D() 202
sram_test_and_clear_bits(163
sram_test_and_clear_bits_D() 163
sram_test_and_clear_bits_D_ind() 164
sram_test_and_clear_bits_ind() 164
sram_test_and_decr() 169
sram_test_and_decr_D() 169
sram_test_and_decr_D_ind() 170
sram_test_and_decr_ind(170
sram_test_and_incr() 167
sram_test_and_incr_D() 167
sram_test_and_incr_D_ind() 168
sram_test_and_incr_ind() 168
sram_test_and_set_bit_pos_D() 203
sram_test_and_set_bits() 161

- sram_test_and_set_bits_D() 161
- sram_test_and_set_bits_D_ind() 162
- sram_test_and_set_bits_ind() 162
- sram_write() 147
- sram_write_D() 147
- sram_write_D_ind() 148
- sram_write_ind() 148
- sram_write_qdesc() 181
- sram_write_qdesc_count() 182
- statements 52
- string literals 34
- supported compilations 20
- switch block packing 63
- synchronization data types 234
 - inp_state_t 234
 - SIGNAL_MASK 234
 - signal_t 234
- synchronization functions 234, 235
 - __signals() 236
 - __wait_for_all() 240
 - __wait_for_any() 240
 - ctx_swap() 238
 - ctx_wait() 239
 - signal_next_ME 245
 - signal_next_ME_this_ctx 246
 - signal_prev_ME 243
 - signal_prev_ME_this_ctx 244
 - signal_same_ME 241
 - signal_same_ME_next_ctx 242
 - signal_test() 237
- T
- thread local 48
- threading model 17
- tips
 - optimizing your code 373
 - things to remember when writing Microengine C code 374
- transfer register
 - read 41
 - write 41
- transfer register modifiers 82
- transfer registers 41
 - arrays of 42
- troubleshooting 378

- program does not fit 378
- program does not run correctly 378

U

- unaligned data access 75
- unaligned get functions 75
- unaligned memory copy functions 80
- unaligned set functions 78
- unsupported ANSI C99 features 71
- user assisted live range analysis 55
- user-guided switch() statement optimization 62

V

- viewing live ranges 57
- viewing local memory usage 48
- virtual register 57
- volatile attribute 43
- volatile registers 43

W

- Workbench
 - running and debugging 30
- write transfer register 41