



Microengine Version 2 (MEv2)

Assembly Language Coding Standards

Revision 1.01g

June 2003





Document History

Version	Description
1.01e August/2002	First external release.
1.01f January 2003	Revised title to IXP2400 and IXP2800 Network Processor
1.01g June 2003	Revised title to Microengine Version 2

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, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © Intel Corporation, June 2003

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

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

Table of Contents

1	Introduction	5
1.1	References	5
2	Capitalization and Identifiers	6
2.1	Lower Case, with exceptions	6
2.2	Descriptive Identifiers	6
3	White Space	8
3.1	4 Column Indentation	8
3.2	Spaces after all commas	8
3.3	Spaces around Operators, Parentheses	8
3.4	Defer Slots	9
3.5	Code Layout	9
4	Comments	9
4.1	Comment Syntax	9
4.2	Block Comments	10
4.2.1	Array Diagram Indexes	10
4.3	Exclude from Comments	10
4.4	Commenting Structured Code	11
4.5	Dead Code	11
5	Pre-Processor Directives	11
5.1	Eighty Columns or Less	11
5.2	Header Guards	12
5.3	#include	13
5.4	#define_eval	13
5.5	#define	13
5.6	Pre-processor and Leading Underscores	14
5.7	#undef	14
5.8	Well Known Preprocessor Constants	14
5.8.1	PARANOIA	14
5.8.2	MICROCODE	14
5.8.3	SIMULATION	14
5.8.4	LITTLE_ENDIAN, BIG_ENDIAN	14
5.9	Conditional Compilation	14
6	Assembler Directives	15
6.1	.reg	15
6.1.1	.reg read/write	15
6.2	.begin/.end vs .local/.endlocal	15
6.3	Use of .begin/.end	15
6.4	.sig	15
6.5	Signals Shared Between Files	16
6.6	Double Signals and mask()	16
6.7	.init	16
6.8	.import_var	16
6.9	.operand_synonym	17
7	Instruction Set	17
7.1	indirect_ref	17
8	Microcode Macros	17
8.1	Where to use Macros	17
8.2	Where NOT to use Macros	17
8.3	Macro API Selection	17
8.4	Macro Names	18
8.5	Microblock Macro Names	19
8.6	Macro Parameters and Arguments	19
8.6.1	Macro Parameter Type and Range Checking	19
8.6.2	Optional Parameters	20
8.7	#macro Parentheses	21
8.8	Macro Template	21
9	File Names	22



10	File Structure	22
10.1	Summary API Block Comment	22
10.2	Header Files	22
10.2.1	.UC non-ANSI-C Compatible Pre-processor Extensions	22
10.2.2	.UC Missing ANSI-C Pre-processor Features	23
10.2.3	Transactor Interpreter (.ind) Pre-processor Limitations	23
10.3	Fast Code Allowing Breakpoints on Exceptions	23
11	Project Configuration and Build	25
11.1	Require Register Declarations	25
11.2	Zero Assembler Warnings	25
12	Project Structure	25
12.1	Sharing Microblocks Across Microcode Projects	25
13	Appendix H: Example Header File	26
14	Appendix UC: Example Assembly File	26

1 Introduction

This standard captures Intel's "Best Known Methods" for programming syntax and style using the Microengine Version 2 hardware with microcode assembly language. Intel/NCG follows these standards so that our customers can best understand, modify, and re-use our source code.

Assembly Language in this context is the source code consumed by the MEv2 assembler – including the MEv2 instruction set, pre-processor directives, assembler directives, and microcode macros – as documented in the Programmer's Reference Manual.

To conform to this standard, the keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, are to be interpreted as described in RFC 2119.¹

We expect an external version of this document to accompany Intel source code shipments to customers. We encourage the development of automated tools to ease compliance with this standard.

Being an assembly language document, the topics are ordered from the bottom-up. We start with the character set and white space, and move up through comments and pre-processor directives, assembler directives, instruction set, microcode macros, file structure and build configuration. But first, some references to other documents.

1.1 References

- IXA Software Building Blocks Developer's Manual. Customer documentation detailing microblocks available on the Intel[®] Internet Exchange Architecture (IXA) Software Development Kit (SDK) 3.1.
- IXA Programmer's Reference Manuals available on the Intel[®] Internet Exchange Architecture (IXA) Software Development Kit (SDK) 3.1)

¹ <http://ietf.org/rfc/rfc2119.txt>

2 Capitalization and Identifiers

2.1 Lower Case, with exceptions

Programmers MUST be consistent in the use of case.² Lower case characters MUST be used for all assembly source code, with these exceptions.

1. Comments SHOULD be normal English sentences, i.e., starting with a capital letter and ending with a period.
2. Pre-processor constant tokens definitions MUST be all capitals.
3. CSR names MUST be all capitals.
4. Labels as macro formal parameters (not arguments) MUST be all capitals.
Note that labels themselves, including Label arguments, MUST be lower case, according to the capitalization rules and the examples shown here.

2.2 Descriptive Identifiers

Identifiers are used to name registers and constants. Identifiers MUST be descriptive and specific. (E.g. \$atm_header, not \$xfer). When identifiers consist of multiple words (or abbreviated words), the words MUST be separated by underscores.

² The assembler is currently insensitive to case, however it is expected that it will be enhanced to be case sensitive to reduce programmer confusion - typically caused by pre-processor token replacement based on all-capital tokens replacing register names which are not all-capital.

```
// Demonstrate all capitalization rules, identifier rules,
// and common white space rules.

#define MAX_LEGAL_INPUT    0x47          // biggest legal lottery number

// lottery_play
//
// Description:
//   Get nothing for something, or go to jail.
//   ...
#define lottery_play(out_cash, in_ticket, LIMIT, BAD_TICKET_HANDLER)
    br=byte[in_ticket, 0, LIMIT, BAD_TICKET_HANDLER]
    immed[out_cash, 0]
#undefm

.reg cash          // global cash balance
immed[cash, 0]     // start with no cash

//-----
.while(1)
    .begin
        .reg contexts
        ...
        local_csr_read[CTX_ENABLES]
        immed[contexts, 0]; immed constant replaced by CSR value
        ...
        lottery_play(cash, cash, MAX_LEGAL_LIMIT, state_prison#)
        ...
    .end
.endw

//-----
state_prison:
    // wind up here when you print your own lottery tickets
```



3 White Space

3.1 4 Column Indentation

Code **MUST** be indented using 4-column granularity. All basic blocks³ **MUST** be indented to reflect structure and flow of control. Eg.

```
#macro indent_demo(out_answer, in_parameter, MAGIC_NUMBER)
.begin
    .reg indent_temp

    immed32(indent_temp, MAGIC_NUMBER)
    .if (in_parameter == 0)
        move(out_answer, indent_temp)
    .else
        move(out_answer, 0)
    .endif
.end
#endm // indent_demo()
```

Note that the Developer's Workbench (DWB) editor sets tab stops to 4 spaces, so tab can be used to secure proper indenting both on the screen and the printed page. However, other tools sometimes use different tab stops and thus will not interoperate with DWB. Requests have been made that DWB be enhanced to expand tabs to the proper number of spaces to address this interoperability issue. Until it is implemented, programmers **MAY** globally replace tabs with spaces after editing with DWB.

3.2 Spaces after all commas

As in English, a comma **MUST** always be followed by a space.

```
sandwich_make(lunch_bag, bread, peanut_butter, jelly)
```

3.3 Spaces around Operators, Parentheses

Arithmetic operators in expressions **MUST** always be surrounded by spaces:

```
#if (a < b)
```

While technically an operator, parentheses are exempt from this rule and **SHOULD** follow the usage in this example. Constant expressions **MUST** be fully parenthesized – they **SHALL NOT** rely on operator precedence for numerical correctness. Eg.

```
#define BIT_PATTERN (3 | (MY_CONSTANT << 2))
```

³ Basic blocks are logical groups of code that reflect naming scope or flow of control. Basic blocks make be delimited by pre-processor directives (e.g. #if/#endif) assembler directives (e.g. .begin/.end) or actual assembly instructions such as jumps and branches.

In several places the instruction set encoding appears to mandate that spaces be excluded around operators. In reality, these constructs are indivisible tokens.⁴ E.g.

```
alu_shf[dest, src_a, b-a, src_b, <<19]      ; no space in b-a or <<19
br!=ctx[0, all_threads#]                    ; no space in br!=ctx
```

3.4 Defer Slots

Code in defer slots of instruction X, is always executed in the cycles immediately following execution of X. Thus, a blank line SHOULD follow the end of the list of instructions in the defer slot, to emphasize the deferred instructions are associated with X. E.g.

```
sram[read, ...], defer[2]
alu[...]
alu[...]

alu[...]
```

As shown above, code in defer slots MUST be at the same level of indenting as the deferring instruction.

3.5 Code Layout

Code SHOULD be written in paragraphs as shown in the appendix. Paragraphs are logical groups of instructions separated by blank lines. Paragraphs begin with a comment describing what the paragraph does.

4 Comments

4.1 Comment Syntax

The assembler supports three choices for comment syntax. As .uc files are processed only by the assembler, .uc files MAY use all three styles.

However, .h files are commonly shared with ANSI-C, C++, and DWB .ind script files, which have compatibility issues:

1. `/* ... */` "slash star" syntax is supported by ANSI-C, C++, and .ind.
2. `/**/` "double-slash" syntax is supported C++ and .ind, but not by ANSI-C.
3. `;"` "semicolon" syntax is supported only by the assembler, as it functions as a command separator in C, C++, and .ind.

Slash star comment syntax MUST be used in .h files, as .h files may be shared with ANSI-C or C++. Double slash and semicolon comment syntax MUST NOT be used in .h files.

Semicolon comment syntax MUST be used only in .uc files. Note that unlike other comments, which are stripped out by the pre-processor, semicolon comments are carried through the assembler and appear in the dis-assembly display. This can be quite helpful during debug.

⁴ Eg. The syntax of the `alu_shf[]` instruction, which requires no spaces between the `<<` or `>>` operators and their parameters `n`: `{<<n, <<indirect, >>n, <<rotn, >>rotn}`, as well as the "b-a" 'operator' and the `br=ctx` instruction.

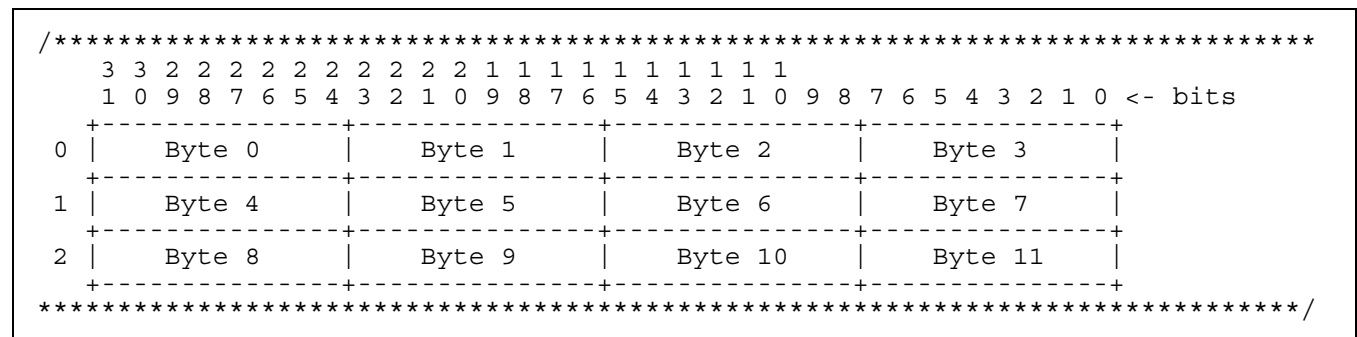
4.2 Block Comments

Block comments which may be copied and pasted to documentation SHOULD use `/* ... */` "slash star" syntax. This eases the editing burden when the comments are changed.

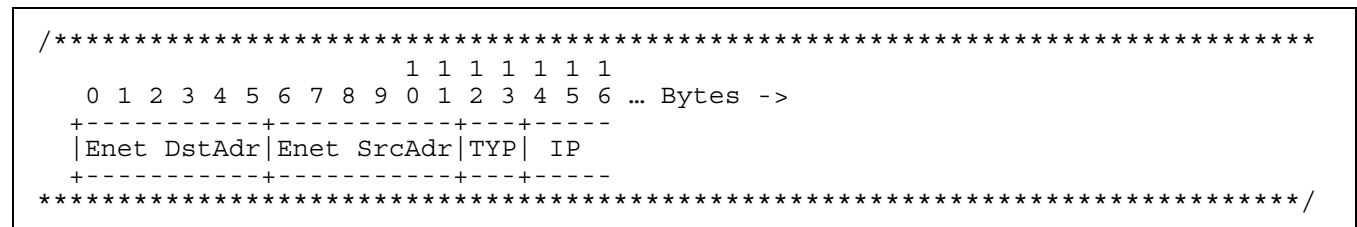
4.2.1 Array Diagram Indexes

When comments include diagrams showing aggregate data types - such as arrays of bits in a register, or bytes in a byte stream, the index of the array MUST be present in the diagram. Either big-endian or little-endian bit and byte order MAY be used, whatever is consistent with the context. The diagram MUST, however, illustrate the order.

In this example illustration of 32-bit data structures in memory, smaller addresses appear toward the top of the figure - as they would appear in a memory dump on the screen. Bit positions are numbered from the right to the left - which illustrates little-endian bit-order.



In the array above, as well as the byte sequence below, bytes are numbered from left to right, which illustrates big-endian byte order.



If the diagrams in code comments are going to make their way into customer documentation, then the diagram conventions in the code and the customer documentation MUST use the same endian conventions.

4.3 Exclude from Comments

Source code comments are customer documentation. The following items MUST NOT appear in source code comments.

1. Exclude reference to personal identifying information, such as employee names, addresses, phone numbers or e-mail addresses.
2. Exclude reference to Intellectual Property - unless it is intended to be disclosed with the source code.
3. Exclude discussion of unpublished errata or errata rumored to be present.
4. Exclude discussion of fixes that may potentially happen in some future hardware or software revision.
5. Exclude references to undocumented features.
6. Comments MAY indicate where and how additional functionality might be added. However, comments MUST NOT imply any commitment that any party will



actually provide that functionality at any time, including vague use of "TBD" or "supplied later".

7. Exclude slang and profanity.

Comments SHOULD include version information identifying the current file version. Comments MAY mention previous file versions to summarize significant customer-visible changes, such as updated APIs. However, exhaustive detailed file histories belong in source code control and SHOULD NOT appear in file comments.

4.4 Commenting Structured Code

When multiple pre-processor or assembly directives are used to structure the logic of code flow, it can be difficult for the reader to associate structure endings with beginnings. (e.g. which .if does this .endif go to?)

Code structure MAY be commented as shown below. Note that the logic of the comment on the 'endif' shows how control entered that block, even if it was the result of an 'else'. Subject to the compatibility rules, any of the three comment syntaxes can be used to clarify code structure.

```
.if (WEATHER == SUNNY)
    yard_work_start()
    ...
.else // rainy weather
    football_game_watch()
    ...
.endif // rainy weather
```

```
#while(CASH_ON_HAND > 0)
    shopping_spree()
#endloop // end while(money left to spend)
```

While these examples are trivial, control structures that are likely to span multiple pages (e.g. 20 lines or greater) MUST be commented.

4.5 Dead Code

There MUST NOT be any commented-out "dead" code. When multiple builds of the same source code are necessary, pre-processor directives SHOULD be used for conditional compilation rather than comments.

5 Pre-Processor Directives

5.1 Eighty Columns or Less

Lines of source code MUST consume 80 columns or less. Note that the assembler pre-processor line continuation feature ('\\') is available to continue long lines. Note also that block comments in the examples are 80 columns wide and can be used as a guide.

The assembler allows comma separated lists to be continued on the next line without the use of the '\\' operator. In the case of macro invocations, the '\\' line continuation character SHALL NOT be used to continue long lines. This is because it is not needed for these lists, and with the current tools it causes the disassembly window to be cluttered with white space.

E.g.

```
// Correct
form_l2a_hdr(*n$index++, prepend_byte_cnt, reassy_index, tx_req_word0,
             buf_desc_word2); write_l2a_hdr to nn ring
```

```
// Incorrect
form_l2a_hdr(*n$index++,
            prepend_byte_cnt,
            reassy_index,
            tx_req_word0,
            buf_desc_word2
            ); write l2a_hdr to nn ring
```

```
// Correct
form_l2a_hdr(*n$index++,
            prepend_byte_cnt,
            reassy_index,
            tx_req_word0,
            buf_desc_word2
            ); write l2a_hdr to nn ring
// comment
// comment
// comment
// comment
// comment
```

5.2 Header Guards

Header (.h) files MUST use header guards to prevent multiple includes of the same file from conflicting. The token for the header guard MUST be constructed by using the base of the file name in all capital letters (i.e. filename sans extension), with the dot in the file name replaced by an underscore, and then the file extension in all capital letters. Therefore:

```
#ifndef <file_basename>_H
#define <file_basename>_H <OPTIONAL version number>

/* Header file contents here ... */

#endif /* <file_basename>_H */
```

The header guard MAY define a version number such that code that includes it has the option to detect a change in versions - perhaps specifying the version that passed evaluation tests. E.g.

```
#include <foo.h>

#if (FOO_H != 42)
    #error included unexpected version (FOO_H) of foo.h
#endif
```

Microcode assembly (.uc) files that are included by other files MUST also use header guards. E.g.

```
#ifndef <file_basename>_UC
#define <file_basename>_UC <optinal version number>

/* Microcode file contents here ... */

#endif /* <file_basename>_UC
```

When used, header guards SHALL occupy the first and last lines of the file.

5.3 #include

Each file SHALL #include only those files needed to build itself.

#include directives SHOULD appear in each file before all code.

As in ANSI-C, there are two variants on parameters to #include - <file.h> and "file.h". The angle brackets form begins each search with the include-path; the double-quotes form begins each search relative to the including file; and then behaves like the angle-bracket form.

The problem with the double-quote form is that when there are multiple levels of nested includes, the 'including file' may not always reside where the programmer expects. Eg. Say there are multiple versions of include directories on the include path, say with updated files appearing earlier in the path. An old version of a file may still be included, even though it appears later on the include path - because it might be included by another file that appears in the same directory as the old include file.

The angle bracket form of #include SHOULD be used, except when the double-quote feature of first searching the directory of the including file is required. For example, when a project includes files that are in a local sub-directory rather than on the search path. E.g.

```
#include "local_sub_directory/header.h"
```

5.4 #define_eval

Both #define and #define_eval create expandable tokens, available for use in other substitutions. As in ANSI-C, #define does not actually evaluate the resulting expression. #define_eval, on the other hand, evaluates it's constant expression at the time the line is processed.

What this means is that expressions created with #define_eval are effectively build-time 'variables' that can be re-defined by subsequent #define_eval's at assembly time. #define_eval is appropriate for use in #for/#while/#repeat loops etc.

Expressions created with #define that incorporate tokens that are created with #define_eval may be 're-defined' by subsequent invocations of #define_eval.

Further, the power of #define_eval to re-define a token without complaint is also a weakness - it allows tokens to be unexpectedly re-defined. For this reason, #define_eval SHOULD be preceded by a check for previous definition.

Due to these potential problems, and the fact that #define_eval is supported only in .UC files, #define_eval MUST not be used when #define will suffice.

Note that the Pre-processor will evaluate constant expressions contained in (), and so that #define_eval is not needed for that purpose.

5.5 #define

There MUST NOT be any "magic number" constants embedded into code. All constants MUST be #defined using self-explanatory upper-case labels. Eg:

```
#define FOOBAR_MASK 0xFF      // 256 possible foobars...
and(answer, input, FOOBAR_MASK)
```

Exceptions to this rule are permitted where economy of expression does not impact clarity, and the constants are unlikely to be modified. E.g. comparison with 0 or 1.

5.6 Pre-processor and Leading Underscores

Pre-processor constants that begin with a double underscore `'__'` SHALL be reserved for use by the tools and framework.

Pre-processor constants that begin with a single underscore `'_'`, SHALL be reserved for application independent libraries and header files.

Applications MUST NOT define preprocessor constants with leading underscores except for explicit interaction with the tools, framework, or libraries. Note that this applies also to header guards.

5.7 #undef

#undef MUST be used to clean the name space when the end of scope of a constant definition is reached. For example:

```
#macro mymac(out_meaning_of_life)
    #define MYMAC_ANSWER 42
    immed32(out_meaning_of_life, MYMAC_ANSWER)
    #undef MYMAC_ANSWER
#endm // mymac()
```

5.8 Well Known Preprocessor Constants

The following well-known preprocessor constants MAY be used only for the functions detailed below. When those functions are being performed, these well-known constants MUST be used.

5.8.1 PARANOIA

Used for all paranoia error checks for debugging. PARANOIA code MAY be left in released source code to aid in maintenance and debugging future modifications. However, session-specific DEBUG code MUST be removed.

5.8.2 MICROCODE

Used to enable common header files for C code and microcode. For example, header files shared between C and assembly that make use of #define_eval MUST protect it with #ifdef MICROCODE.

5.8.3 SIMULATION

Used to enable common header files for simulation and hardware. As simulation and hardware environments SHOULD be kept as similar as possible, the need for distinguishing them with SIMULATION SHOULD be minimized.

5.8.4 LITTLE_ENDIAN, BIG_ENDIAN

LITTLE_ENDIAN is global setting that specified how packet numeric fields are accessed. Default is BIG_ENDIAN.

5.9 Conditional Compilation

#if or #ifdef support conditional compilation so that the same source code can support multiple build options. However, released code MUST NOT include any untested or un-supported options – even if conditionally compiled out by default.

During pre-release development, it can be helpful to use conditional compilation to manage code changes. Constructs such as `#ifdef NOT_YET_TESTED`, or `#ifndef ABOUT_TO_DELETE` MAY be used for transient code, but along with `#if 0`, they MUST not appear in released code.

6 Assembler Directives

6.1 .reg

All registers MUST be declared using `.reg`. This SHALL be enforced by enabling the project's assembler build setting *require register declarations*.

For maintainability, register declarations SHOULD be grouped near their use, as shown in the examples in Appendix UC.

6.1.1 .reg read/write

Transfer register declarations MUST limit the use of the registers to READ or WRITE when the intent of the register is only for reading or only for writing. E.g.

```
.reg read $atm_header           // read only
.reg write $cell_padding       // write only
.reg $semaphore                // read and write
```

6.2 .begin/.end vs .local/.endlocal

When registers are explicitly scoped, the `.begin` and `.end` directives SHALL be used, and the `.local/.endlocal` directives SHALL NOT be used.

6.3 Use of .begin/.end

Use of globally declared registers (those outside any `.begin/.end` scope) SHOULD be minimized. When used, they SHOULD be declared one register per line, followed by a comment explaining what the register is for. In the microblock environment, global registers SHALL be used only for microblocks that live on a single microengine or microengine thread, e.g. Queue Manager.

Note that the assembler tracks the minimal "live range" for registers, so `.begin/.end` scoping is not necessary to create correct or optimal code. However, it allows the assembler to identify bugs when registers are used outside their intended scope. It also promotes a readable and maintainable code structure.

6.4 .sig

All signals MUST be defined using the `.sig` assembler directive. This SHALL be enforced by assembling all code with the *Require register declarations* build option. Assigning constant values to signals using `.addr` directive SHOULD be avoided.

For maintainability, signal declarations SHOULD be grouped near their use, as shown in the examples in Appendix UC.

6.5 Signals Shared Between Files

When signal definitions are shared between files, the signal number MUST be #defined as a constant in a shared header, and that constant definition consumed by the files that need to know about the signal, as shown in this example.

```
/* Define shared signal in .h file */
#define MY_SIGNAL_NUMBER <n>
```

```
// Consume "global" signal in .uc file
// while limiting its scope
.begin
    .sig foo
    .addr foo MY_SIGNAL_NUMBER
.end
```

6.6 Double Signals and mask()

Read/Modify/Write operations can result in two signals – the 1st for the read, and the 2nd for the write. This implementation detail is largely hidden by the assembler, but is exposed when the programmer manipulates signal numbers directly. When manipulating signal bits, the mask() directive MUST be used to hide the possible generation of multiple signals. E.g.

```
sram[test_and_inc, ...]sig_done[sig_foo] // generates 2 signals
alu[wakeup_sigs, wakeup_sigs, or, (mask(sig_foo)), <&(sig_foo)]
ctx_arb, defer[1]
local_csr_wr[ACT_CTX_WAKEUP, wakeup_sigs]
```

In this example mask(sig_foo) is needed rather than a 1, because the atomic operation generates 2 signal bits.

6.7 .init

The .init directive SHOULD be used to initialize registers and local memory when the initial constants are known at build time.

6.8 .import_var

.import_var is available only when running on hardware. Such variables MUST also have default values available to support SIMULATION. E.g.

In dl_system.h:

```
/* define simulation default for IPV4_ROUTE_TABLE_BASE */

#ifdef SIMULATION
    #define IPV4_ROUTE_TABLE_BASE 0x10000 // only for simulation
#endif
```

In microblock specific header:

```
/* on hardware, set IPV4_ROUTE_TABLE_BASE at load time */

#ifndef SIMULATION
    .import_var IPV4_ROUTE_TABLE_BASE /* will be patched by XScale code */
#endif
```



The tool chain defines the following well-known import variables, `__CHIP_ID`, `__CHIP_REVISION`, `__UENGINE_ID`, as detailed in [PRM].

6.9 `.operand_synonym`

This deprecated operator MUST NOT be used.

7 Instruction Set

7.1 `indirect_ref`

The `max_nn` token SHOULD be used when the `indirect_ref` token is used to change a reference count, as described in [PRM]. When `max_nn` cannot be used, e.g. in macros, `max` should be used.

Comments MUST be used to describe use of `indirect_ref`.

8 Microcode Macros

8.1 Where to use Macros

Microcode assembly language macros SHOULD be used for the following reasons:

1. Support code re-use and ease of programming.
2. Hide complexity, particularly hardware dependencies.
3. Code readability and maintainability.

8.2 Where NOT to use Macros

It is possible to over-use the power of macros by attempting to hide too much detail. The high-level flow of control SHOULD remain easy to follow, modify, and maintain:

1. Major decisions in program flow SHOULD be explicit (outside the macro).
2. Branches within the macro SHOULD generally be low-level conditionals, or branches within a fully-implemented algorithm.
3. Thread parallelism, interaction with other threads and units SHOULD NOT be hidden.

8.3 Macro API Selection

The easiest to use macros are entirely self-contained, and have general APIs that allow reuse under a wide variety of conditions. Macro APIs SHOULD be chosen so that everything that controls how a macro expands is specified in its API. Macros SHOULD NOT access global variables that do not appear as part of the macro API.

An exception MAY be made for macros that hide global state from their callers. The global state is private to the macro or macro sub-system and the caller need not be burdened with knowing about it. E.g.

```
// @last_element does not appear as a parameter in this macro's API

#macro tbuf_alloc[out_tbuf_element]
    .reg global @last_element
    .init @last_element 0

    alu[out_tbuf_element, ..., @last_element]
    ...
#endm
```

Macro expansion depending on global settings outside the macro API SHALL be limited to the list of well-known global #defines (above).

When functions can't be fully contained within a macro, the macro API SHOULD highlight it. For example, the use of an exception handler label in a macro API.

8.4 Macro Names

Macro names SHALL follow the same syntax rules as register identifiers. I.e. macro names MUST consist of lower case words and abbreviations separated by underscores.

The macro name itself MUST consist of the following ordered sequence of character groups:

1. partition prefix. Identification of the sub-system partition. This will identify a partitioned API group that represents a communication type, protocol type, or bus interface type, generally a noun.
2. sub-partition. Optional. This can further identify an operation as working on specific functionality, generally a noun or adjective.
3. verb. This identifies action being performed by the macro.

I.e. <partition_prefix>_<sub-partition>_<verb>()

Examples:

```
ipv4_proc(...)          // process an ip datagram
ipv4_verify(...)        // verify an ip header
ipv4_cksum_verify(...)  // verify an ip header checksum
```

Macro source will consist of external (API) level macros, as well as internal utility macros. The intent is that the API level macros will maintain a stable API, and that the internal utility macros may exist just for internal code maintainability or may not be appropriate to export. Internal utility macro names MUST begin with a leading underscore. External API macro names MUST NOT begin with a leading underscore. Eg.

```
#macro example_external_demo()
    ...
    _example_internal_demo()
    ...
#endm // example_external_demo()
```

8.5 Microblock Macro Names

Microblocks MUST have the following externally visible macros

```
<microblock_name>_init()
```

```
<microblock_name>()
```

E.g. `ipfilter_init()` and `ipfilter()`

Any macros defined *internally* by a microblock MUST follow the naming convention `_<microblock_name>_xxx()` .e.g `_qm_handle_enqueue()`. This allows easy identification of macros belonging to a microblock in the workbench macro listing.

The format of the macro MUST be per the examples in Appendix UC.

8.6 Macro Parameters and Arguments

Macro parameters are specified in macro definitions, while the term macro argument refers to macro invocation.

Parameters and arguments SHOULD be ordered as follows:

1. outputs. Results, or registers modified by the macro, including write transfer registers.
2. output/inputs. Registers that are read and modified by the macro.
3. inputs. Registers read by the macro (not modified).
4. signals and constant inputs. Tokens representing immediate values used by the macro, including signal names.
5. labels. Symbolic addresses for exception handlers.
6. optional parameters. Tokens which may or may not be supplied.

Exceptions may be made for APIs with uniform handles. In these cases some of the macros may treat a parameter as an input, while others may treat it as an output.

In the macro definition, macro input/output GPR parameters SHOULD begin with `out_`, `io_`, or `in_`.⁵

CONSTANT parameters and arguments MUST be capitalized.

8.6.1 Macro Parameter Type and Range Checking

Input parameters can be invoked with either variable (register) or CONSTANT arguments. When a macro fails to build for constant arguments, the build SHOULD fail with an informative message.

When constant arguments have a limited legal range, macros SHOULD check their range at build time.

⁵ In the long run we'd prefer that the assembly tools implement macros as in-line functions with the ability to **declare** when parameters are inputs or outputs, such that the assembler can enforce the declarations. But this `out_/_io/_in` naming convention at least helps humans perform that checking.

Example definition:

```
#macro dinner_bill_pay(io_balance, in_cost, ACCOUNT_NO, \
    OVERDRAFT_HANDLER, BAD_PIN_HANDLER)

    #if isnum(in_cost)
        #if (in_cost > MAX_DINNER_BUDGET)
            #error dinner_bill_pay(in_cost) invalid "in_cost"
        #endif
    #endif

    #if (ACCOUNT_NO > MAX_ACCOUNT_NO)
        #error dinner_bill_pay(..., ACCOUNT_NO, ...) invalid "ACCOUNT_NO"
    #endif

    //...
#endm
```

Example invocation:

```
dinner_bill_pay(account_balance, price, MY_ACCOUNT, wash_dishes#,
    pin_re_enter#)
...
wash_dishes#:
    ...
pin_re_enter#:
    ...
```

Label parameters **MUST** be capitalized, as they are technically constants. Label arguments, on the other hand, **MUST** be lower case, according to the capitalization rules and the examples shown here.

Use of labels as macro parameters **SHOULD** be minimized, as they can make the flow of control difficult to read. As such, they **SHOULD** be used primarily for the addresses of exception handlers.

Macros **MUST NOT** access global state (e.g. registers) that can be passed in through the macro API. All parameters passed to a macro **SHOULD** be used in the macro. Exceptions (E.g. an API with a common handle for multiple macros) **MUST** be documented in the block header for the macro.

8.6.2 Optional Parameters

The assembler allows overloading macro definitions so that the same macro name may correspond to different macros with different number of parameters. This allows programmers to build macros with optional parameters. When optional parameters are used, they **MUST** be last in the parameter list. Eg.

```
#macro label_read(out_answer, in_address)
    _label_read(out_answer, in_address, no_option)
#endm

#macro label_read(out_answer, in_address, in_option)
    _label_read(out_answer, in_address, in_option)
#endm

#macro __label_read(out_answer, in_address, in_option)
    sram[read, ..., in_address, ...], in_option
    ...
#endm
```

8.7 #macro Parentheses

Parentheses - ()' - MUST be used for all #macro definitions and invocations. (This is to help distinguish them from native assembly language statements and array subscripting syntax which use []'s', particularly in the black-and-white context of paper printouts.)

8.8 Macro Template

The following template MUST be filled in for each externally shared macro definition. The OPTIONAL fields MAY be filled in when deemed appropriate, but if used they MUST appear in the order shown using the labels shown.

```
// <macro_name>
//
//   Description:
//
//   Parameters:
//     Outputs: <brief description>
//     In/Outs: <brief description>
//     Inputs: <brief description>
//     Constants: <brief description>
//     Labels: <brief description>
//
//   Size: instruction count (OPTIONAL)
//
//   Performance: <cycles - possibly multiple cases or estimates> (RECOMMENDED)
//
//   Register Use: <type and quantity of registers used> (OPTIONAL)
//
//   Signal Use: (OPTIONAL)
//
//   Side effects: <global assumptions, shared state use ($nn, RAM)> (OPTIONAL)
//
//   See also: <reference to other libs or doc> (OPTIONAL)
//
//   Example Usage: <use of macro in a line of code> (OPTIONAL)
//
#macro <macro_name>(args...) >
.begin
    .reg <local register names...>
    <code> ; comments
.end
#endm    // <macro_name>
```



The "Example Usage" field is REQUIRED for public APIs, but is OPTIONAL for private APIs.

If there are numeric limits on the values of macro parameters (eg. 0-1 or 0-63), they MUST be documented in the Parameter section of the block header. The range of constant parameters SHOULD be checked with pre-processor directives. The range of variable parameters MAY be checked at run-time within PARANOIA code.

9 File Names

File base names MUST follow the same syntax rules as identifiers. I.e. File names MUST consist of lower case words and abbreviations separated by underscores.

Header files shared between the assembler, C, C++, .ind files SHALL use the .h extension.

Microcode SHALL appear only in files using the .uc extension.

Reusable macro sub-systems and libraries called <partition_prefix> SHOULD reside in a file named <partition_prefix>.uc.

Files containing microblocks MUST have names beginning with a prefix denoting the block. (e.g. all dispatch loop files SHOULD start with dl_*, all Queue Manager files SHOULD start with qm_. This makes it easy to find files in the file tab in the workbench. All files of a block are automatically grouped together)

Large microblocks MAY be split into multiple files. When multiple files are used to implement a single microblock, they MUST use file names structured as in this example for the 'ipfilter' microblock. E.g. ipfilter.uc and filter.h MAY be joined by ipfilter_init.uc and ipfilter_util.uc.

10 File Structure

10.1 Summary API Block Comment

Files which export macros MUST have a summary of the exported API in block comment near the top of the file.

10.2 Header Files

Header (.h) files defining system parameters may be included by either assembly language microcode or the Micro-C compiler. They may also be included by assembly, C or C++ source files that run on the Intel[®] XScale core, or by .ind files processed by the Transactor's C-interpreter. When the same file is used in several of these realms, it MUST employ only features that are compatible in each realm. Generally ANSI-C pre-processor constructs with ANSI-C style comments are universal. Some traps to avoid:

10.2.1 .UC non-ANSI-C Compatible Pre-processor Extensions

These constructs do not work in C, C++, or .ind files

- #macro, #endm
- #define_eval
- #repeat, #while, #endloop
- #for, #endloop



10.2.2 .UC Missing ANSI-C Pre-processor Features

The Assembler pre-processor does not recognize some common ANSI-C pre-processor idioms:

```
#define max(a, b) (a > b ? a : b)
```

10.2.3 Transactor Interpreter (.ind) Pre-processor Limitations

The transactor interpreter does not handle some common ANSI-C pre-processor idioms:

- #if/#elif/#else/#endif⁶
- #include
- path command works for simulation, but not hardware mode⁷

10.3 Fast Code Allowing Breakpoints on Exceptions

Code MUST be structured such that it simultaneously meets cycle budget while still allowing modification and debugging. One method for optimizing cycle budgets is to remove the exception code from the fast path. E.g.

```
while1#:  
    ...  
    br_bset[status_reg, ERROR_BIT, bad_bit_handler#]  
    // fast path code executes here after 1 cycle  
    ...  
br[while1#]                ; branch to top of nominal loop  
  
// Exception handlers at bottom of file  
//-----  
bad_bit_handler#:  
    // exception case code pays for branch latency  
    br[while1#]            ; return to fast path
```

This structure allows the nominal fast path to test for an exception without any branches or aborted cycles from empty defer shadows. The exception case pays the branch cost. It also allows the programmer to place a breakpoint on the exception handler without it firing in the normal case. A similar example allows visibility into when polling loops fail:

```
while1#:  
    ...  
    check_ring_status#:  
        br_inp_state[ring_full, check_ring_status#] ; no space for breakpoint  
        // nominal "ring available" code follows      ;keep looping  
    ...  
br[while1#]
```

The same code MAY be written to allow better visibility into the exception case without any performance penalty on the nominal case:

⁶ "transactor interpreter support for #if/#elif/#endif"

⁷ "WB cmd interpreter rejects PATH command"

```

while1#:
    ...
    check_ring_status#:
        br_inp_state[ring_full, ring_not_available#]           ; jump to exception case
        // nominal "ring available code follows"
    ...

br[while1#]
//-----
// ring not available exception handler
ring_not_status#:
    // exception case code goes here (if any)
    // breakpoints that fire on exception can go here
    br[check_ring_status#]

```

11 Project Configuration and Build

11.1 Require Register Declarations

All new projects MUST have the "Require Register Declarations" feature enabled in the project build settings.

11.2 Zero Assembler Warnings

DWB and the assembler support multiple verbosity levels for assembler warnings. If the build produces multiple build warnings, then it is difficult for customers to find new warnings caused by their changes. For this reason, all released code MUST NOT produce any assembler warnings at verbosity level 3, and SHOULD NOT produce any assembler warnings at verbosity level 4.

New code MUST NOT be added to address assembler warnings, rather assembler directives such as `.set` and `.use SHALL` be used.

12 Project Structure

12.1 Sharing Microblocks Across Microcode Projects

Since the source for a microblock may be shared across multiple projects the source code MUST NOT contain references to other microblocks

For example, for a scratch ring used by the POS receive block to send input requests to the Queue Manager block, the source code for the Queue Manager MUST only contain references to `QM_ENQUEUE_RING_IN` and the POS Receive block MUST only contain references to `POS_RING_OUT`. The actual values for `QM_ENQUEUE_RING_IN` and `POS_RING_OUT` MUST be defined to the same value in the system header file `"dl_system.h"` which is project specific.

This allows the same queue manager code to be reused in a different project where it may accept enqueue requests from a different microblock e.g. Ipv4.

13 Appendix H: Example Header File

```
#ifndef DL_SYSTEM_H
#define DL_SYSTEM_H

/*****
                                Intel Proprietary

Copyright (c) 1998-2002 By Intel Corporation. All rights reserved.
No part of this program or publication may be reproduced, transmitted,
transcribed, stored in a retrieval system, or translated into any language
or computer language in any form or by any means, electronic, mechanical,
magnetic, optical, chemical, manual, or otherwise, without the prior
written permission of:
                                Intel Corporation
                                2200 Mission College Blvd.
                                Santa Clara, CA 95052-8119
*****/

/*
 * dl_system.h
 * System-wide Definitions used at Dispatch Loop Level (1-line abstract)
 */

/*
 * <Top level contents of this file, including exported APIs,
 * assumptions about how the file is used etc.>
 */

/* Global Definitions */

#define MEANING_OF_LIFE 42          /* <reason for global definition> */

#endif    /* DL_SYSTEM_H */
```

14 Appendix UC: Example Assembly File

```
#ifndef MICROBLOCK_STYLE_UC
#define MICROBLOCK_STYLE_UC 1

/*****
                                Intel Proprietary

Copyright (c) 1998-2002 By Intel Corporation. All rights reserved.
No part of this program or publication may be reproduced, transmitted,
transcribed, stored in a retrieval system, or translated into any language
or computer language in any form or by any means, electronic, mechanical,
magnetic, optical, chemical, manual, or otherwise, without the prior
written permission of:
                                Intel Corporation
                                2200 Mission College Blvd.
                                Santa Clara, CA 95052-8119
*****/

// microblock_style.uc
// Demonstrate Assembly Language Coding Standards (1 line file abstract)

// Exported APIs:
// macro_typical_show(io_sum, INCREMENT, EXCEPTION_HANDLER)
```



```
// long_arg_list_show(out_answer, CONSTANT_A, CONSTANT_B, CONSTANT_C,
//   CONSTANT_D, CONSTANT_E)

// Assumptions:
// This file may be included by others (and thus uses header guards)

//-----
// Included files: (include only those files necessary to build this file)
#include "dl_system.h"          // Dispatch Loop level system definitions

//-----
// File-wide global definitions

//   The Size of meta data (32) in SRAM, expressed in Long words.
//   (ie <2 will give bytes).
#define META_SIZE_LW           0x8

//-----
//   Global Registers (if any)

.reg example_global_register    //   Tell briefly what this register is for.

//-----
//   Global Signals (if any)

.sig example_global_signal      //   Tell briefly what this signal is for.

//-----
// <macro definition>

// <macro_name>
//
//   Description:
//
//   Parameters:
//     Outputs: <brief description>
//     In/Outs: <brief description>
//     Inputs: <brief description>
//     Constants: <brief description>
//     Labels: <brief description>
//
//   Size: instruction count (OPTIONAL)
//
//   Performance: <cycles - possibly multiple cases or estimates> (RECOMMENDED)
//
//   Register Use: <type and quantity of registers used> (OPTIONAL)
//
//   Signal Use: (OPTIONAL)
//
//   Side effects: <global assumptions, shared state use ($nn, RAM)> (OPTIONAL)
//
//   See also: <reference to other libs or doc> (OPTIONAL)
//
//   Example Usage: <use of macro in a line of code> (OPTIONAL)
//
#macro macro_typical_show(io_sum, in_count, INCREMENT, EXCEPTION_HANDLER)
.begin
    .reg _local_temp, address, $output_data
    .sig _local_sig
    immed[_local_temp, 0x42]
    immed[address, 0]
```

```

#ifdef _LOCAL_CONSTANT
    #error    "_LOCAL_CONSTANT is already defined" (_LOCAL_CONSTANT)
#endif
#define _LOCAL_CONSTANT 0x4

alu[$output_data, --, b, _local_temp]
sram[write, $output_data, address, count, 1], ctx_swap[_local_sig], defer[2]
alu[_local_temp, _local_temp, +, INCREMENT];          _local_temp +=
INCREMENT
alu[_local_temp, _local_temp, +, in_count];           _local_temp += in_count

alu[io_sum, io_sum, +, _local_temp]
alu[--, io_sum, -, _LOCAL_CONSTANT]
bge[EXCEPTION_HANDLER]

//    cleanup name space.
#undef _LOCAL_CONSTANT
.end
#endm // <macro_name>

#macro long_arg_list_show(out_answer, CONSTANT_A, CONSTANT_B, CONSTANT_C, \
    CONSTANT_D, CONSTANT_E)
    alu[out_answer, out_answer, xor, CONSTANT_A]
    alu[out_answer, out_answer, xor, CONSTANT_B]
    alu[out_answer, out_answer, xor, CONSTANT_C]
    alu[out_answer, out_answer, xor, CONSTANT_D]
    alu[out_answer, out_answer, xor, CONSTANT_E]
#endm

//-----
//    Main entry point -- Code begins execution here.
//-----
main#:
    .reg count
    // run time initialization
    immed[example_global_register, 0]
    immed[count, 0]
//-----
while1#:
    macro_typical_show(example_global_register, count, 3, exception_handler#)
    long_arg_list_show(example_global_register, 0x00000055, 0x000000AA,
0x00000088, 0x00000011, 0x42)
    alu[count, count, +, 1]
    br_bset[count, 0, bad_bit_handler#]
    br[while1#]
//-----
bad_bit_handler#:
    br[while1#]
//-----
exception_handler#:
    immed[example_global_register, 0]
    br[while1#]
//-----
#endif /* MICROBLOCK_STYLE_UC */

```