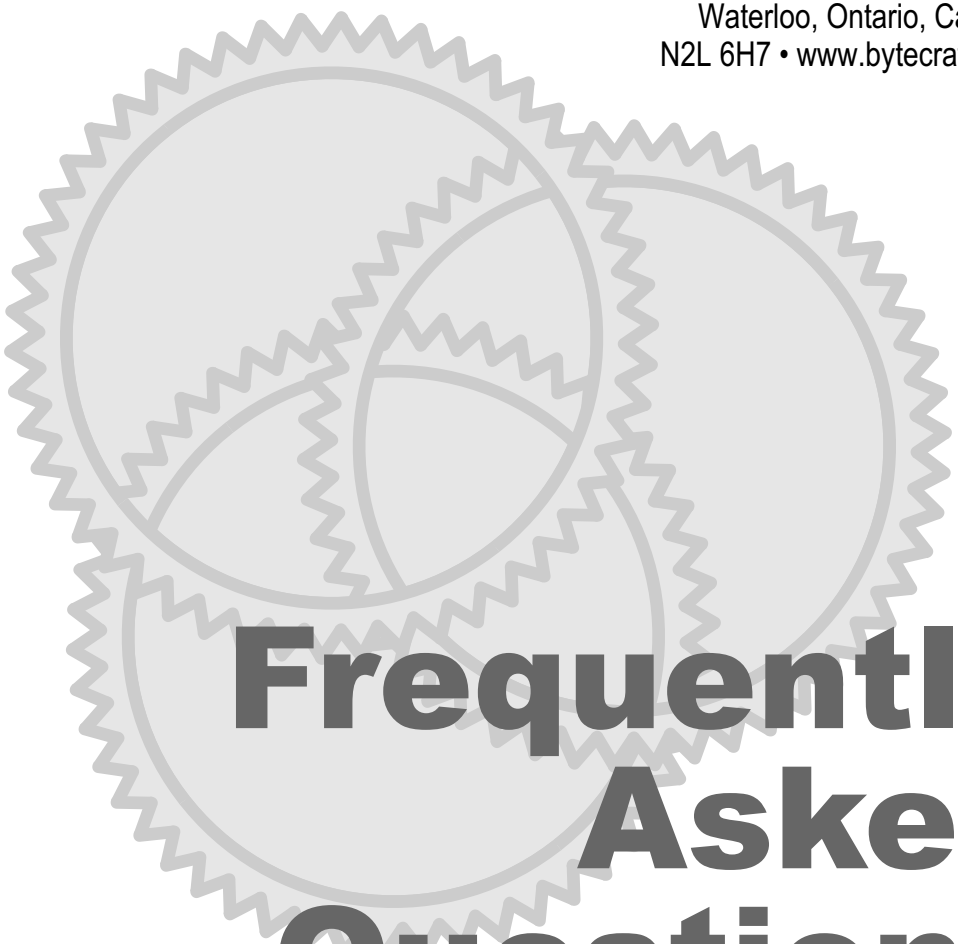
eTPUC

Code Development System

Optimizing C Compiler and Development Tools
for the Freescale eTPU

Byte Craft Limited
A2-490 Dutton Drive
Waterloo, Ontario, Canada
N2L 6H7 • www.bytecraft.com



Frequently Asked Questions

Legal Notices

Trademarks:

- PC-DOS and IBM are trademarks of International Business Machines Inc.
- Windows, Windows NT, and MS-DOS are registered trademarks or trademarks of Microsoft Corporation.
- Pentium is a trademark of Intel Corporation.
- Motorola is a registered trademark of Motorola Inc. In May 2004, Motorola SPS changed its name to Freescale Inc. Freescale is a registered trademark of Freescale Inc.
- PowerPC is a registered trademark of International Business Machines Inc.
- Other named and terms used herein are trademarks of their respective holders.

Disclaimer

While every attempt is made to ensure the accuracy and completeness of the information in this document, some errors may exist. Byte Craft Limited cannot accept responsibility of any kind for a customer's losses due to their use of this document.

Byte Craft Limited reserves the right to make changes without notice in the products or software described or contained herein in order to improve design and/or performance.

Byte Craft Limited assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright or make work right to these products or processes, and makes no representation or warranty that these products are free from patent, copyright or make work right infringement, unless otherwise specified.

Applications that are described herein for any of these products are for illustrative purposes only. Byte Craft Limited makes no representation or warranty that such applications are suitable for the specified use without further testing or modification.

Warranty

Byte Craft Limited warrants the physical storage medium and documentation to be free of defects in materials and workmanship for a period of thirty days from date of purchase. In the event of written notification within the warranty period of defects in material or workmanship, Byte Craft Limited will replace the storage medium or documentation. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, special, incidental, consequential or other claims.

Byte Craft Limited specifically disclaims all other warranties, expressed or implied, including but

not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the storage medium and documentation, and without limiting operation of the program license with respect to any particular application, use or purpose.

In no event shall Byte Craft Limited be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.

Use in Life Support must be expressly authorized

Byte Craft Limited products are not for use as critical components in life support devices or systems without express written approval of an officer of Byte Craft Limited. As used herein:

- Life support devices or systems are devices or systems which support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
- A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

This warranty shall be governed by the laws of the Province of Ontario in Canada.

License Statement

This software is protected by both Canadian copyright law and international treaty provisions. Therefore, you must treat this software as you would a book, with the following exception: Byte Craft Limited authorizes you to make archival copies of the software for the sole purpose of backing up our software to protect your investment from loss.

This software may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of it being used at one location while it is being used at another.

This software can not be used by two different people or on two different computers at the same time.

Byte Craft Limited grants you, the licensed owner of a Byte Craft Limited Code Development System, the right to incorporate library routines into your programs. You may distribute your programs containing Code Development System library routines in executable form without restriction or fee, but you may not give away or sell any part of the Code Development System library source code.

Support

We provide technical support to all registered Byte Craft Limited Customers.

All Byte Craft Limited products include free updates for one year after purchase, or until the next major release, whichever is *longer*. You are also entitled to a free upgrade to any major release within one year of the original purchase. After one year, upgrades are available to registered customers only, at a reduced price.

You must register your Byte Craft Limited product, to receive technical support and information about any new versions of this software. To register, return the registration card to Byte Craft Limited or complete the form on our website at: http://www.bytecraft.com/regprod.html.

Byte Craft Limited

Byte Craft Sdn Bhd in Malaysia is not associated with Byte Craft Limited in Canada. Byte Craft Sdn Bhd is a systems development and consultancy company in Kuala Lumpur serving South East Asia.

BCL Technologies is not associated with Byte Craft Limited in Canada. BCL Technologies sells document management and voice recognition software.

Revision History

0.5	**July 2004**	**Initial Draft**
0.6	September 2004	Additional content
0.7	October 2004	Additional content
0.8	January 2006	Additional content
0.9	July 2006	Additional content
1.0	June 2007	Corrections
1.0p	July 2007	PDF; additions

Table of Contents

Examples

Tables

Welcome

Byte Craft Limited has always been committed to programmers working within the C language. We specialize in making C available for targets with limited resources or specialized architectures. We seek to comply with relevant C standards within the capabilities of the target part.

eTPU is a truly specialized architecture. To say desigining for eTPU was a challenge is an understatement. Programming eTPU in assembly falls under the "anything is possible" category. We've done it, and we've seen it done, but not often. To get code of any efficiency requires an assembler with capabilities near to those of a compiler.

Optimized hardware does add to the complexity of the eTPU. That's why Byte Craft Limited has been involved with the eTPU from the beginning, contributing machine-generated language expertise. No matter how intricate the eTPU might be, the engineers designing with it need to meet the same kind of deadlines.

In order to program a distinctly-different type of controller with C, we've specified some eTPU_C-specific idioms for programmers to use.

This document collects techniques we've learned along the way.

> *Presentation*
>
> Due to the presence of listing file examples, some pages may not be presented properly (lines may stretch off the edge of the page). Please accept our apologies.

1. The Product

Questions about eTPU_C features

1. Isn't hand-optimized code more efficient?

   I've found an instance where optimization should have happened but didn't. Why is this so?

   There is no technical reason why compiled code on the eTPU cannot be as tight as hand-optimized code. The ISO C standard for embedded systems overcomes previous C language limitations related to embedded.

   There are three things that the eTPU_C compiler can do very well that hand optimization finds difficult.

   1. Computers are very good at accounting.
   2. Programmers tricks, once added to the knowledge base in the eTPU_C compiler, remains available for everyone's applications forever. The compiler is capable of combining these tricks into combinations that we have not anticipated.
   3. Most of the detail of the instruction set and eTPU architecture is embedded in the compiler. Application developers can rely on this and focus on application details.

   In the eTPU_C compiler, we have placed in the code generator rules that allow for sub-command re-ordering. What we found was there are only a few cases where this can happen without un-wanted side effects. We can implement complex rules only as we gather more experience identifying what should be allowed. The problem is the same as for hand coding, except we only need to get the details right once.

   Optimizers have three parameters to work with: code size, ram requirements and execution cycles.

   - Optimization occurs when one or more of the three is reduced but not at the expense of the remaining parameters.
   - Compression occurs when there is an exchange between parameters where one or more are reduced at the expense of the remaining parameters.

   Another way to look at this: optimizers will not make a bubble sort into a quick sort, but they will make the best bubble sort possible.

2. What should I know about linking?

   Linking was most useful in the age of limited computing resources, on self-hosting systems. Since our customers are cross-developing, and compiling resources are effectively unlimited (given the typical sizes of the embedded executables they create), linking has declined in importance.

   We advocate *Absolute Code Mode*, a single compile-and-link step. Absolute Code Mode simply compiles the main C module to an executable, as opposed to an object file.

In fact, choosing to use object files can raise some interesting consequences:

- Our compilers perform application-level optimization, to allow the compiler to pare the executable down as much as possible. This requires that **#pragma** directives from the device header file be present (or **#include**d) in the linker command file, and that preprocessor directives can also appear there.
- When compiling to object, the compiler will raise an error when **#include**d files are missing. It will do this even when the **#include** is discounted by preprocessor statements (conditional compilation).

3. What is Absolute Code Mode?

   Absolute Code Mode in Byte Craft Limited compilers generates executable code from the compiler without a linking step. Absolute Code Mode still allows you to select object code from libraries and include it as part of an application.

   The compiler will read in any functions that are referenced in the eTPU_C program and are available in a library file. Simply **#include** the library's header file at the top of a program module, or (if there is no header) **#include** the library itself at the end of the main program.

   For example, consider a program that uses spark, fuel, cam, and crank **ETPU_function**s. These statements will select the referenced eTPU functions from `engine.lib` and assign them to individual eTPU function numbers.

   Example 1: engine.lib members

```
#pragma ETPU_function spark @ 2;
#pragma ETPU_function fuel  @ 3;
#pragma ETPU_function cam   @ 4;
#pragma ETPU_function crank @ 5;

#include <engine.h>
```

   When creating library files, enclose the code in **#pragma library** and **#pragma endlibrary** statements. These signal the compiler to read into the program only those functions that are referenced in the main program. Compile the library sources to an object file, and rename the object file with a `.lib` extension.

   The significance of the `.lib` extension: `.lib` files are automatically included every time a `.h` file is **#include**d in a source program. In all cases, library files should be renamed from `.obj` to `.lib`.

4. I need to combine one eTPU executable into another eTPU executable. How can I do this?

   Use the BClink directive **ETPUIMAGE**. During this linking, the specified executable will be

linked in to the final executable. eTPU_C will perform checks to prevent conflicts between the two executables.

1. Export the first executable with the following eTPU_C host interface macro in the first linker command file.

```
#pragma write msc, ( ::ETPUcode );
```

2. Import the first executable during linking for the second executable, using the **ETPUIMAGE** directive instead of an **OBJECT** or **LIBRARY** directive.

   Export a new executable and MISC value for the host compiler's use.

```
// Output a code image of both links
#pragma write q, (const u32 etpu_code[] = { ::ETPUcode32};);
// Create the combined MISC value
#pragma write q, ( #define MISC ::ETPUmisc );

// Read the code image of the first link and combine it with the second
link
ETPUIMAGE= etpu_a_CPU.msc
```

5. How do I generate a S-record file of the executable?

   To generate an S-record executable file, do the following:

   1. Open your project in BCLIDE.
   2. Choose *Project*|*Properties*, and click on the *Compiler Properties* tab.
   3. In the *Hex Dump File* dropdown box, choose either "S1", "S2", or "S3" for the appropriate S-record output.
   4. Ensure that the command line option **+delf** is *not present* in the *Additional Options* below.
   5. Click *OK*, and compile your project as usual.

   If you're not working within BCLIDE, add **+dS1** or **+dS2** or **+dS3** to the compiler command line.

   This will generate an S-record file for the entire executable.

   The file will be zero-based and a complete executable, as it is generated from eTPU's point of view. To extract any pieces out of the file or alter it in other ways (such as changing the base address of the file), you'll need to use an external tool.

6. How do I interpret the reports the compiler generates?

eTPU_C generates some reports in the listing file that describe variables and **`ETPU_functions`**.

```
   00F8 clocal                              signed int16        0200  0203
```

This type of report appears in the *RAM usage map*. It reports on the location that a variable occupies in RAM (some values also represent registers or parameter offsets) and its type.

It describes the range of instructions in the program for which this declaration is in scope and valid. Local variables allocated to the same locations are not overlapping so long as their scope ranges are different.

```
     0           one  0200  0210  (4 words)  SRAM =  0  Local RAM =  8
```

Each **`ETPU_function`** has an entry in the *eTPU Function Summary*. The function number at left describes its entries' position in the entry table. The record describes the function's lowest and highest ROM bytes and the number of 32-bit instruction words it uses.

Its parameter RAM and local RAM usage is also reported. Local RAM usage is for the entire function execution: memory allocated to local storage needs to equal the largest local RAM specification given here. The amount of parameter RAM needed can be used in host interface programs by expanding the host information macro `::ETPUfunctionframeram(`*x*`)`, where *x* is either the function number or name given in these reports.

7. Is an eTPU_C available for Linux or Mac?

   eTPU_C is a Windows executable. It should run under a Windows emulator or environment on Linux or Mac. However, we have little experience running it in these environments and cannot offer technical support for platform-specific issues. eTPU_C relies only on core functions in Windows that should be well supported.

   Issues previously experienced include slow speed of execution, display and interactivity problems, and outright crashes.

2. Programming Technique

Technique

Questions about programming technique with eTPU_C

8. How should I structure my code?

   There are a few rules to follow:

   - Functions that might not get linked in to the final executable should be put in libraries (modules with **`#pragma library`** and **`#pragma endlibrary`** directives at the beginning and end, respectively). This instructs the compiler to omit generating code for them if they're not invoked.
     Order within the link is important too. Libraries containing any such functions must be linked in *after* the main program.
   - Note that the above doesn't change the C requirement that functions be prototyped before they are invoked. Your library should have a standard header file with function prototypes.
   - Both headers and program modules should have **`#ifdef/#define/#endif`** protection, to prevent compilation errors:

   ```
   #ifndef __MODULE_C
   #define __MODULE_C

   /* ... */

   #endif /* __MODULE_C */
   ```

   If you've written single-file libraries without this protection, it might cause a problem when compiling using Absolute Code Mode.

9. What causes threads to start?

   Threads are started by eTPU events (host/link service or match/capture events). Execution starts at the **`ETPU_function`** associated with a channel.

   In eTPU_C, the `if()..else if()..else` structure "tests" the channel conditions, and causes the appropriate code to execute. The compiler translates the `if()` structure into the channel condition encoding, and uses that as position information for the thread entry point in that **`ETPU_function`**'s portion of the entry table.

   Though all the tests available to you in this context are valid (they aren't misleading), some of the tests check what caused the thread to start, and others check parametric information.

   - Threads are actually invoked by Host Service Requests (`hsr` set to something other

than 0), Link Service Requests from another thread, or Match or Transition events.
- Pin state and flag states (`Flag0` and `Flag1`) are testable but are never directly responsible for a thread starting.
- There is always an implied test of `hsr`. A thread that doesn't explicitly test `hsr` will still only run when

```
hsr == 0
```

10. My threads aren't starting as expected. What's wrong?

    Ensure that you give as much information to the compiler in the `if()..else if()..else` expressions as possible. Explicitly test `lsr`, `m1/m2`, `pin`, and `flag0/flag1`. Most importantly, add the `hsr` test explicitly, as the first term evaluated in an `if()` expression. Remember that `hsr` is always tested whether or not it's present in the thread expression. If it's not there, it is tested against a `hsr` value of 0.

    Also note whether the **`#pragma ETPU_function`** directive specified the **`standard`** or **`alternate`** condition code encodings.

11. How do I access parameters coherently?

    Add the following declarations to your program:

```
#define _coherentread(a_source,a_dest,b_source,b_dest) \
   {register_diob diob; register_p p; diob = a_source; \
    p = b_source;  b_dest = p; a_dest = diob;};

#define _coherentwrite(a_dest,a_source,b_dest,b_source)\
   {register_diob diob; register_p p; diob = a_source; \
    p = b_source; a_dest = diob; b_dest = p; };

register_diob diob;
register_p    p;

#define CoherentWrite(a,av,b,bv) diob = av; p = bv; NOP(); a = diob; b = p;
```

    where `a_dest` and `b_dest` are the variables, and `a_source` and `b_source` are values to assign to `a_dest` and `b_dest`.

    There are a few restrictions for using this macro:

    - Only the `b` arguments can have 32 bit variables (because they use the `p` register) The `a` arguments in these macros must be 24 bits. The `b` arguments may be 24 or 32 bits.
    - Expressions for `b` arguments cannot use anything that requires the `diob` register (arrays, pointers in general). There are no restrictions on the `a` expressions.

12. I need to place variables manually. How can I do this?

You can place variables in their declaration:

```
type identifier @ address;
```

Simply append an `@` symbol and a valid address. Since eTPU_C can calculate constant expressions at compile time, you can use relative addresses:

```
void xyz (int32 a,b,c)
{
  int16 myarg1 @ &a;
  int16 myarg2 @ &a + 2;
  int8  myarg3 @ &b;
  int8  myarg4 @ &b + 1;

  /* ... */
}
```

Important:

eTPU_C does not take such explicitly-placed variables into account when allocating memory for other variables. Check the listing file to ensure variables do not conflict.

Because eTPU is a multi-processor system (host and one or two engines), placing variables can avoid conflicts that semaphores or parameter coherency cannot resolve.

13. How can I recover the the contents of the Capture registers after ERT_A/B have been written over by programming a match?

    Invoke:

    ```
    chan = chan;
    ```

    The compiler will not optimize this out, as there are side effects.

    Example 2: Recovering the Capture register values

    When a thread starts, ERT_A/B are loaded with the values of the Capture registers. This code reloads those values after using ERT_A/B for other purposes.

```
                                        /* Thread starts, ERT_A/B =
Capture */

                                        erta = 0x555;
0200 015F253D   alu erta = #0x0555.
0204                                    ertb = 0xAAA;
```

```
0204 02AF36D9   alu ertb = #0x0AAA.
0208                                    channel.ERWA = 0b0;
                                        channel.ERWB = 0b0;

0208 7FF94F43   alu chan = chan ,ccs;   chan = chan;
                    chan write_erta,
                    write_ertb.
```

14. I'm having problems when the host and eTPU are using variables in the same PRAM word. Both are performing read-modify-write sequences on their own words, but the eTPU is overwriting the host's update. Is this a bug?

    No, but it will require you to use semaphores to arbitrate access to the PRAM location.

15. I've performed an **MDU** division in my code. How can I access the remainder?

    Access the `MACH` register directly:

```
Result = Value1 / Value2;
Remainder = mach;
```

16. Does eTPU_C support signed divide?

    No. Signed divide is not supported at this time.

17. How do I use eTPU semaphores?

    The short answer: assign a number from 0 to 3 to `channel.SMPR`, and loop until `channel.SMPR` tests true. Perform the sensitive work to be protected by the semaphore, and assign -1 to `channel.SMPR`.

    Alternatively, use these macros in `eTPUC_common.h`:

```
// Semaphore operations
#define IsSemaphoreLocked()    (channel.SMPR == 1)
#define LockSemaphore(num)     (channel.SMPR = num)
#define FreeSemaphore()        (channel.SMPR = -1)
```

    Semaphores are intended for communication between eTPU engines, when two (or potentially more) of them appear in an eTPU subsystem. A channel can set a semaphore, (non-destructively) test whether the other engine set the same semaphore, and clear the semaphore it set.

Simply use the above statements in two **ETPU_function**s that differ in their channel assignment.

There's little reason to use them within one channel context: eTPU threads are run-to-completion. The end of a thread clears the semaphore, but leaving it that long is not recommended as it could significantly delay the work of the other execution engine.

> *Important:*
>
> While you can set one of 4 different semaphore values, you don't need to test which number was set. Simply test `channel.SMPR`: if it's true, you've set the same semaphore as the other engine, and your work might interfere with its ETPU_function's work. Wait until `channel.SMPR` is clear before proceeding.

Example 3: Semaphores

This example demonstrates semaphore operations. The two functions below would be called by ETPU_functions running on different engines. Since they both use semaphore 2, the protected sections will not execute at the same time.

```
                                        void one(void)
                                        {
                                          do
                                            {
                                              channel.SMPR = 2;
                                            }
0200 F7A0101F   if smlck==0 jump 0200,    while (channel.SMPR == 0);
                    noflush.
0204 FFFFF7DB   ram lock_g2.

                                          /* Do protected work */

                                          channel.SMPR = -1;

                                          /* Continue */
0208 FFFFCCF9   return,noflush.         }
020C FFEFF7FB   ram free_g.

                                        void theother(void)
                                        {
                                          do
                                            {
                                              channel.SMPR = 2;
                                            }
0210 F7A0109F   if smlck==0 jump 0210,    while (channel.SMPR == 0);
                    noflush.
0214 FFFFF7DB   ram lock_g2.
```

```
                                        /* Do protected work */

                                        channel.SMPR = -1;

                                        /* Continue */
0218 FFFFCCF9   return,noflush.       }
021C FFEFF7FB   ram free_g.
```

18. How do I access the angle clock?

    The angle clock hardware allows microcode to watch the turning of a gear, and act at angles or rotations.

    The TPR register, type `register_tpr` is specified as a structure:

```
struct tpr_struct {
  int TICKS   : 10;
  int TPR10   : 1;
  int HOLD    : 1;
  int IPH     : 1;
  int MISSCNT : 2;
  int LAST    : 1;
  } ;
```

    Also, `AddAngle()` and `SubAngle()` macros (that perform modulo angle operations) are available in `etpuc_util.h`.

3. Language

This section gives answers about the C language as implemented in eTPU_C.

Questions about eTPU_C Language

19.

Table 1: Types and sizes

20. I want to assign an **`ETPU_function`** to more than one channel. This function has `static` variables declared in it. Will all channels share this variable in common? If not, how will the function address them?

    Each channel, when it runs a shared **`ETPU_function`**, can have its own function frame based at its own `CPBA` setting. This allocation contains parameters and local static variables.

21. How do I make better use of **`ETPU_function`** parameters from inside a subordinate C function?

    `ETPU_function` parameters don't just pertain to the code in the **`ETPU_function`** body; they are relevant to everything that happens in the thread. It would be undesirable to pass these parameters around, and very difficult to get a globally-valid reference.

    This item describes a way to use a matched set of **`ETPU_function`** parameters and a special global pointer to get access to the parameters in a C function.

    Accessing C pointers

    1. Declare a global structure of the eTPU function parameters, and declare the `chan_base` register as a pointer to the structure. The parameters of the current can then be referenced by name by the called function. This does not incur code or execution time overhead.

```
struct _xyz_arg {
  int chan_arg1,
      chan_arg2,
      chan_arg3;
} register_chan_base *pxyz_arg;
```

    `register_chan_base` is the processor-specific type for the `chan_base` register; any declared variable of this type refers to `chan_base`.

    2. Define an **`ETPU_function`** with `struct _xyz_arg` as the parameters. This is a slight

inconvenience, but has the same effect as if the structure members were individual parameters.

```
int i;

#pragma ETPU_function xyz @ 3;

void xyz (struct _xyz_arg args)
{
  if (hsr == 3)
    {
      i = args.chan_arg2;
    }
}
```

Within the ETPU_function, code accesses the parameters indirectly:

```
    i = args.chan_arg2;
```

3. C functions access the ETPU_function prameters through the global `chan_base` pointer; it always points to the current ETPU_function parameter list.

```
void func ( void )
{
  int x;
  x = pxyz_arg -> chan_arg3;
}
```

Dereferencing this pointer by the "->" operator is the same as direct access to the eTPU function parameters, and doesn't require the address calculation at run-time in software. There is no code penalty to access parameters this way. The generated access code is identical to the parameter access in the eTPU function.

22. C includes a `register` data type. Does eTPU_C allow register allocations?

How do I access the P register from C?

eTPU_C does offer register-like types, with a few catches. You can use these types to access hardware registers directly, if necessary.

The traditional `register` alerts the compiler to allocate very fast storage (ie., a processor register) for a variable. It's a suggestion, and not mandatory. `register` might be useful on a system with lots of similar general-purpose registers. On eTPU, using `register` types is less of an advantage.

eTPU_C provides several specific `register_xx` types, one for each microcode-accessible register. Identifiers declared with them guarantee access to the named register.

Registers declared as global variables cannot be used to hold local variables or intermediate results.

`register_xx` variables actually point to a register, and therefore have no address value suitable for a pointer.

23. Can I use **`goto`** to jump between sections of the `if()..else if()..else` in an eTPU function?

    In general, no. Remember that the bodies of this top-local-scope statement are individual eTPU threads. The compiler treats them as run-to-completion code, and may optimize them differently.

    To share programming between more than one thread within an eTPU function, create a C function and call it from all relevant threads.

24. How do I enable or disable match events during a thread?

    How do I set or clear the ME flag in the Entry Point?

    How do I set the PP (parameter preload) flag in an Entry Point?

    Use the intrinsic functions `enable_match()` and `disable_match()` to do this.

    These functions generate no code. Look for the results in the listing file, in the entry point report following each part of the **`ETPU_function`** `if()/else if()/else` structure.

    Example 4: Enabling matches during a thread

    Note: some additional entry report lines deleted for space

```
                                        void handler(void)
                                        {
                                          if(pin == 1)
                                            {
                                              enable_match();
0204 4FFFFFFF   nop.                          NOP();
0208 6FFFFFFF   end.                        }
0014 40 81        00 S0A P01 ME 0204  HSR 0     lsr 1  m1 1  m2 1  pin x
flag1 x  flag0 0

0000             Thread Local RAM size
                                          else if(pin == 0)
                                            {
                                              disable_match();
020C 4FFFFFFF   nop.                          NOP();
0210 6FFFFFFF   end.                        }
0018 00 83        00 S0C P01 MD 020C  HSR 0     lsr 0  m1 0  m2 1  pin 0
flag1 x  flag0 0

0000             Thread Local RAM size
```

```
                                        else
                                          {
```

By default, match events are enabled.

To set or clear the PP flag, use either of the `preload_p01()` or `preload_p23()` intrinsics.

25. How do I re-read the Capture registers into ERT1/2?

    How do I read the Match registers into ERT1/2?

    To refresh ERT1/2 with the value of the Capture registers, re-assign the `chan` register:

    ```
    chan = chan;
    ```

    To load ERT1/2 with the value of the Match registers, use the `read_match()` intrinsic.

    ```
    read_match();
    ```

26. I declared a `static local` variable with the same name as a function parameter. The compiler didn't catch the problem. Is this a bug?

    It's not a bug; it is a problem with the C language itself. The local shadows the parameter.

27. How should I initialize globals and static locals?

    It's possible to use a Host Service Request to perform initialization, but we don't feel it's the best way.

    We recommend using the host interface macros to create an initialization routine to run on the host.

    Here is one way:

    Example 5: Host Initialization

    This example shows a basic initialization process.

```
0200                                    #pragma write c, (

                                        int pram_location;
                                        int pram_top;
                                        #define init_pram(where, towhere)
(pram_location = where, pram_top = towhere)
                                        #define alloc_pram(howmuch)
(pram_location += howmuch)
                                        #define exceeds_pram(howmuch)
((pram_location + howmuch ) > pram_top ? 1 : 0)
```

```
                                        /* ETPUglobalimage expansion omitted
due to size: using
                                           const char etpu_image[]; */

                                        void initialize_eTPU(void)
                                        {
                                          /* initialize SCM */
                                          /* alternative, use
ETPUcodeimagesize */
                                          for(int i; i < 4 ; i++)
                                            outb(ETPU_SCM_BASE+i,
etpu_image[i]);

                                          /* initialize eTPU globals */
                                        #define __etpu_globals(offset,
valtype, value) outb(ETPU_PRAM_BASE+offset, (valtype)value);

                                          ::ETPUglobals ;

                                          /* initialize channel 1 */
                                          /* ::ETPUfunctionnumber(handler)
*/
                                          CxCFS(1, 1);

                                          init_pram();

                                          if(exceeds_pram(::ETPUfunctionfram
eram(handler))) error(OF_SOME_KIND);
                                          CxCBPA(1,
alloc_pram(::ETPUfunctionframeram(handler)));

                                        #define __etpu_staticinit(offset,
value) outb(CBPA_BASE(1)+offset, value);

                                          ::ETPUstaticinit(1);


                                        }
                                        );
```

The generated host interface file (here edited for length) looks like this:

```
 int24 pram_location;
 int24 pram_top;
 #define init_pram(where, towhere) (pram_location = where, pram_top =
towhere)
 #define alloc_pram(howmuch) (pram_location += howmuch)
 #define exceeds_pram(howmuch) ((pram_location + howmuch ) > pram_top ? 1 :
0)
 const char etpu_image[] = { 0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
```

```
                                0xC0,0x85,0x40,0x81,0xC0,0x85,0xC0,0x85,
                                0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
                                0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
                                0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
                                0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
                                0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
                                0xC0,0x85,0xC0,0x85,0xC0,0x85,0xC0,0x85,
                                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                                0xFF,0xDF,0xCC,0xF9,0x1C,0xBF,0xAF,0xBE,
                                0xCF,0xFF,0xF1,0xFF,0xFF,0xC0,0x10,0x1F,
                                0x6F,0xFF,0xFF,0xFF,0x4F,0xFF,0xFF,0xFF,
                                0x6F,0xFF,0xFF,0xFF,0xFF,0xDF,0xCC,0xF9,
                                0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
                                 };
 void initialize_eTPU(void)
 {
   for(int24 i; i < 4 ; i++)
     outb(ETPU_SCM_BASE+i, etpu_image[i]);
 #define __etpu_globals(offset, valtype, value) outb(ETPU_PRAM_BASE+offset,
(valtype)value);
   __etpu_globals(CC                                ,cc_reg    ,0x0000)
__etpu_globals(global1                            ,sint24    ,0x0001)
__etpu_globals(global2                            ,sint24    ,0x0005)
 ;
   CxCFS(1, 1);
   init_pram();
   if(exceeds_pram(0x0000 )) error(OF_SOME_KIND);
   CxCBPA(1, alloc_pram(0x0000 ));
 #define __etpu_staticinit(offset, value) outb(CBPA_BASE(1)+offset, value);
   ;
 }
```

28. The documentation talks about a device header file and **`#pragma memory`** declarations, but I don't see any file with such directives. Where are they?

    The compiler has defaults (see section ,) that apply to most eTPU programs, so device header files aren't strictly necessary to compile eTPU programs.

    In some situations, you might want to create a header file with configuration statements that alter the compiler's default settings for code memory and parameter RAM. This file is typically described as the "device header file". It must be available to the compiler and BClink (if used).

4. Implementation

Implementation

Questions about eTPU_C program implementation

29. What defaults are built in to the compiler?

    The default SCM space for generated code starts at 0x200. This allows etpu functions 0-7 to be defined in the entry tables.

    To allow more entries, this base rom space must be moved. This is done with

    ```
    #pragma memory ROM [size] @ base_address;
    ```

    For example:

    ```
    #pragma memory ROM [0x8000-0x400] @ 0x400;
    ```

    will allow for space for 16 entries in the eTPU function entry tables.

    The default SPRAM space runs from 0x0000 to 0x0400. Allocations for global variables start at 0 and ascend. Locals and parameters not allocated from registers begin at the upper bound of SPRAM and descend.

    You can change this setting with the **`#pragma memory RAM`** directive.

30. eTPU is word-oriented. How does this affect eTPU_C?

    It doesn't. Internally, eTPU_C deals with bytes; this makes it easier for C programmers to work with eTPU_C. During code generation, eTPU_C makes a conversion to sub-word calculations, using different strategies to get at different bytes.

    The only caveat you need to remember is this: code for operations on byte-sized values can be particularly expensive in terms of instructions. Whether it's warranted is a matter for the designer to decide.

    Example 6: Byte-wide operations

    This example demonstrates how eTPU_C handles byte-wide values.

    In this example, eTPU_C "parses" the target address, and uses the appropriate byte of P to host the assigned value.

```
0000                                    int8 a;
0001                                    int8 * z;

                                        /* with pointers */
                                        z = &a;
```

```
0200 9FFF7B00   alu p = 0 ;
                    ram 0001 = p23_0.
0204 1C8F0FFE   alu a = #0x23.            *z = 0x23;
0208 9FEFFF00   ram diob = 0001.
020C FFEFF8D9   ram p31_0 = (diob).
0210 080BFBDA   alu nil = diob &
                    #0x000002,ccs.
0214 F0E0119F   if z==0 jump 0230,
                    noflush.
0218 080BFBBA   alu nil = diob &
                    #0x000001,ccs.
021C F0C0115F   if z==0 jump 0228,flush.

0220 3FF9AFF4   alu p31_24 = a .
0224 F7C0121F   jump 0240,flush.
0228 3FF9BFF4   alu p23_16 = a .
022C F7C0121F   jump 0240,flush.

0234 3FF9CFF4   alu p15_8 = a .
0238 F7C0121F   jump 0240,flush.
023C 3FF9DFF4   alu p7_0 = a .
0240 FFFFF8D9   ram (diob) = p31_0.
```

Thinking of SPRAM as a byte-oriented memory, eTPU is big-endian: the MSB will appear in the (effectively) lowest address space.

31. eTPU_C is allocating my variables all over the place. Some are allocated out of order! What's wrong?

    The compiler has specific rules about allocating variables, temporary locations, and registers.

    Globals are allocated in low memory locations (from 0x0000 up). Locals are allocated in high locations (from 0x0400 down). These locations are reported in the listing file, at the left-hand side of declarations.

    If possible, the compiler will use a register instead of a memory location. Registers are assigned special pseudo-locations inside the compiler. In the listing file, the values appearing for declarations allocated from variables represent these internal pseudo-locations, even if they appear out of order or overlap other variables.

32. I've declared an `int8` as a global, and another `int8` as a `static` local. They are both allocated at the same location. What's wrong?

    Variable allocations are all over the place: some are out of order, and some are at impossible/unimplemented locations. What's wrong?

    The listing file displays truncated memory allocation information obtained from the compiler's internal tables. Two addresses that appear exactly the same or from

unimplemented locations represent virtual allocations made by the compiler, into registers or other address spaces used internally. The code will address the variables correctly.

33. An expression that uses division isn't working properly. What's wrong?

    Due to implementation limitations, signed divides are not available.

34. The compiler has optimized a local variable into a register. I'd prefer it take a memory location. How can I change this?

    The compiler has correctly optimized the variable into a register because it could do so. Declare the variable as a `static` variable, and eTPU_C will be forced to allocate a location.

35. eTPU_C is giving "RAM allocated out of default RAM space" warnings. What's wrong?

    This warning can occur when **`#pragma memory RAM`** or **`#pragma memory LOCAL`** directives declare RAM that doesn't start on a 4-byte boundary. Variable declarations will try to allocate locations partly outside of declared RAM space

36. How are structure bit fields dealt with?

    As efficiently as possible. Bit fields are packed within 32-bit boundaries. Packed bit fields can have a dramatic impact on the amount of code generated.

    Example 7: Bit Fields

    This example declares two `structs`, one with less than 32 bits of bit fields, and one with more.

```
                                          struct {
0000 01 00                                  unsigned int shortElement : 1;
/* 1 bit in size */
0000 11 01                                  unsigned int longestElement :
17; /* 17 bits */
0002 06 02                                  unsigned int longElement : 6;
/* 7 bits in size */
0000 0004                                 } myBitField;

                                          struct {
0000 0F 00                                  unsigned int fifteenbits : 15;
0001 04 07                                  int fourbits : 4;
0004 11 00                                  unsigned int seventeenbits: 17;
0004 0008                                 } myLongerBitField;



0200 9FEFFB00   ram p23_0 = 0001.         myBitField.shortElement = 1;
0204 0800DBA2   alu p7_0 = p7_0 |
                    #0x000001,ccs.
0208 9FFFFB00   ram 0001 = p23_0.
```

```
020C CFEFF000   ram p31_0 = 0000.          myBitField.longElement = 0x55;
0210 1FF03FE4   alu p = #0x03FFFF.
0214 3B180FF2   alu a = p & p ,ccs.
0218 19597382   alu p = a | #0x540000,
                    ccs.
021C CFFFF000   ram 0000 = p31_0.

0220 9FEFFB01   ram p23_0 = 0005.          myLongerBitField.fourbits = -2;
0224 09F87FF2   alu p = p & #0xFF7FFF,
                    ccs.
0228 1BE87392   alu p = p & #0xF8FFFF,
                    ccs.
022C 000FF418   alu diob = #0xFF0000.
0230 3B787FF0   alu p = p | diob ,ccs.
```

37. The ASH WARE simulator has given a warning about an **MDU** subinstruction paired with a **CCS** subinstruction. It calls the subinstruction choice "puzzling". What's wrong?

    This is effectively a "don't care" operation. The **MDU** flags are always preserved. The compiler uses **CCS** for all math operations, except where explicitly disabled by optimization.

    See also section 2, Programming Technique.

5. Host Interfacing

Questions about host interfacing

38. How do I cause a global exception?

    The eTPU requires a value of 0x02 in the CIRC instruction field.

    In C, CIRC is part of the `channel` structure:

    ```
    channel.CIRC = 0x02
    ```

    Alternatively, use a macro from `eTPUC_common.h`.

```
#define SetGlobalException()        (channel.CIRC = 2)
```

39. What other ways can I signal the host?

    There are two other exception types: channel interrupts and data transfer interrupts. In those devices without DMA hardware, the two are indistinguishable. Assign values 0 or 1 to `channel.CIRC`, or use these macros from `eTPUC_common.h`.

```
// Channel control macros
#define SetChannelInterrupt()       (channel.CIRC = 0)
#define SetDataTransferInterrupt()  (channel.CIRC = 1)
```

40. How should the host communicate with the eTPU?

    There are a few ways:

    - You can pass information through `static` locals. The host sets these itself, during initialization.
      Use the `::ETPUlocation` macro to identify the offset of the `static` in the function frame. Rather than assigning the value in `::ETPUstaticinit` which comes from the eTPU_C program, simply set your own initial value.
      For more information on host interface macros, see section 8, eTPU Host Interface Variables.

41. What else do I need to know?

    In order for the host to manage the eTPU_C program, it needs information gleaned from the eTPU_C program at compile time.

    See section 8, eTPU Host Interface Variables for a way to extract information from the eTPU_C program at compile time and make it available to the host's program at compile time.

6. Moving from Assembly

If you've programmed eTPU in assembly, this section includes information on how the compiler exposes eTPU functionality through the C language.

Moving from Assembly

42. How do I perform a **`read_mer`** or **`read_mer12`** operation?

    Use the `read_match()` intrinsic function.

43. What registers are available for inline assembly?

    The `DIOB`, `A`, `P` in all of its forms (`p31_0`, `p31_16`, and so on) are all available for inline assembly essentially without any issues.

    The `B` register is used as a temporary location in expression processing. It is usually freed by the end of a C statement, allowing inline assembly.

    The `MACH` and `MACL` are sometimes used by the compiler in C functions that don't involve multiply and divide operations as storage for local variables. Register usage in inline assembly is NOT tracked by the compiler; diagnose any conflicts through the listing file.

    The `C`, `D` and `SR` are used for local variables in **`ETPU_function`** threads. Any of these registers may be declared as global variables in an application; if so declared, the compiler will not use registers for local variables.

7. Troubleshooting

1. I'm getting the error

   ```
   OVERWRITING PREVIOUS ADDRESS CONTENTS xxxx Conflict in location of entry
   table in ROM
   ```

   What's wrong?

   There are two main possibilities:

   - An ETPU_function number is duplicated between two ETPU_functions. Check the **`#pragma ETPU_function`** declarations in your program.
   - There are more than 8 ETPU_functions and the entry address table has not been moved. See the **`#pragma entryaddr`** directive.

8. eTPU Host Interface Variables

8.1. eTPU_C information

eTPU_C generates useful information intended for the CPU host. eTPU_C expands host interface macros within **`#pragma write`** commands to communicate this information to the host CPU compiler. The values of the macros are described below.

Many eTPU_C host interface macros expand to C macro calls themselves. The macros called are not defined by eTPU_C; this convention allows you to process information generated by eTPU_C with the host CPU compiler. In many cases, the macro calls generated will be sequential, suitable for use in a constant array. If you choose to create an array, the C macro you write can supply the comma separator. Remember that C arrays may have a trailing comma after the last constant value; you need no special handling of the last macro expansion.

Numerous host interface macros accept either an ETPU_function name or number interchangeably. If you specify a function name, number, or parameter name that is not valid, eTPU_C will write an error message to the host interface file but not stop compiling.

Some macros include type information. The typing information is expressed in a single keyword, as follows:

Table 2: Type keywords for Host Interface Macros

Type	Description
uint8	Unsigned 8-bit integer.
uint16	Unsigned 16-bit integer.
uint24	Unsigned 24-bit integer.
uint32	Unsigned 32-bit integer.
sint8	Signed 8-bit integer.
sint16	Signed 16-bit integer.
sint24	Signed 24-bit integer.
sint32	Signed 32-bit integer.
ufract24	Unsigned 24-bit fixed-point value.
sfract24	Signed 24-bit fixed-point value.

eTPU_C expands the following macros within **`#pragma write`** text:

> *Note:*
>
> If macros accept parameters, no space between the macro identifier and the opening parenthesis is permitted.

8.1.1. ::ETPUcode, ::ETPUcode32

```
::ETPUcode

::ETPUcode32

::ETPUcode(size)

::ETPUcode32(size)
```

A constant array image of the eTPU program binary code. `::ETPUcode` represents the program in `char`s, while `::ETPUcode32` uses 32-bit *unsigned* `int`s.

Given without an *`size`*, the macros expand to the entire program. Given with an *`size`*, eTPU_C will expand the program image from offset 0 to the size given. The *`size`* may be a host interface macro like `::ETPUmaxrom`.

The following declaration will generate a constant array carrying the whole eTPU application for the host interface code.

```
#pragma write c,(const char etpu_program[] = { ::ETPUcode }; );
```

For a short eTPU program, the resulting `::ETPUcode` array looks like this:

```
const char etpu_program[] = { 0x9F,0xEF,0xFB,0x00,0x9D,0xF8,0x7A,0x00,
                              0x9F,0xEF,0xFB,0x00,0x9E,0xF8,0x7A,0x00,
                              0xFF,0xDF,0xCC,0xF9,0xFF,0xC0,0x00,0x1F,
                              0x6F,0xFF,0xFF,0xFF,0xFF,0xDF,0xCC,0xF9,
                              0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

and so on, continuing to the end of an array the size of SCM. The `::ETPUcode32` array looks like this:

```
const char etpu_program32[] = { 0x9FEFFB00,0x9DF87A00,0x9FEFFB00,0x9EF87A00,
                                0xFFDFCCF9,0xFFC0001F,0x6FFFFFFF,0xFFDFCCF9,
                                0x00000000,0x00000000,0x00000000,0x00000000,
```

and so on. To emit just the program image, use the following:

```
#pragma write c, (const char etpu_program[] = { ::ETPUcode(::ETPUmaxrom) }; );
```

8.1.2. ::ETPUcodeimagesize

```
::ETPUcodeimagesize
```

An integer size in bytes of the eTPU code image. This image includes all program code as well as unused program locations. Contrast with `::ETPUmaxrom`.

You can use this to determine the size of `::ETPUcode`, and divide it by 4 to determine the size of `::ETPUcode32`.

8.1.3. ::ETPUentry

```
::ETPUentry(function number)

::ETPUentry(name)
```

A list of entry points implementing the eTPU function, expressed as offsets into SCM. Each element in the list is in turn a C macro in the form

```
__etpu_entry(entry_number, start_address)
```

`entry_number` is a number from 0 to 31. The addresses emitted by this macro are entry points to the instructions of the thread.

The following declaration will emit a table of the entry points for an eTPU function `myfunction()`.

```
#pragma write c, (
#define __etpu_entry(entry, offset) offset,

const void * myfunction_bases[] = { ::ETPUentry(myfunction)  };
);
```

Note that an extra value need not be appended to the array to terminate the comma-separated list of values generated by the macros.

8.1.4. ::ETPUentrybase

```
::ETPUentrybase
```

The base address of the first active entry table.

8.1.5. ::ETPUentrytables

```
::ETPUentrytables
```

A comma-delimited list of all the entry table base addresses.

8.1.6. ::ETPUentrytype

```
::ETPUentrytype(function number)

::ETPUentrytype(name)
```

The channel condition encoding format, standard or alternate, of an eTPU function (by name or number). A `0` value indicates *standard entry encoding*, while a `1` indicates the *alternate entry encoding*.

The following declaration will test the encoding of `myfunction()`:

```
#pragma write c, (
#if ::ETPUentrytype(myfunction) == 0
#warning myfunction() channel condition encoding is STANDARD
#else
#warning myfunction() channel condition encoding is ALTERNATE
#endif /* ::ETPUentrytype */
);
```

The output in *basename*`_CPU.c` appears as follows:

```
#if 0 == 0
#warning myfunction() channel condition encoding is STANDARD
#else
#warning myfunction() channel condition encoding is ALTERNATE
#endif /* ::ETPUentrytype */
```

8.1.7. ::ETPUfilename

```
::ETPUfilename(filename including path)
```

This macro does not expand directly to any value. When expanded, it causes a side effect for the host interface file it's being written to. The host interface file is created with the file name given as the macro argument.

By default, host interface files are emitted with filenames of *name*`_CPU.`*letter*, where *letter* is specified in the **`#pragma write`** statement that performs the write. `::ETPUfilename` changes this name to a user-specified filename.

Example 8: ::ETPUfilename usage

```
#pragma write h, (/* Code for <name>_CPU.h */);

#pragma write h, (::ETPUfilename(first_header.h)); //write h messages to
first_header.h
#pragma write h, (/* Code for first_header.h */);

#pragma write h, (::ETPUfilename(second_header.h)); //write h messages to
second_header.h
#pragma write h, (/* Code for second_header.h */);
```

8.1.8. ::ETPUfunction

```
::ETPUfunction
```

A list of all the eTPU function names and numbers. Each element in the list is in turn a C macro in the form

```
__etpu_function(name, number, engine)
```

where *name* is the eTPU function name, *number* is the function's number, and *engine* a `1` or `2` depending upon which engine will run the function. The following declaration will emit a report of the function list:

```
#pragma write c, (
#define __etpu_function(name, number, engine) name engine:number

/* eTPU function list
::ETPUfunction
   end of list */
);
```

The output in *basename*`_CPU.c` appears as follows:

```
#define __etpu_function(name, number, engine) name engine:number

/* eTPU function list
__etpu_function(myfunction, 1, 1)
   end of list */
```

Preprocessing the code will expand the function entries into records describing each function.

8.1.9. ::ETPUfunctionframeram

```
::ETPUfunctionframeram(function number)

::ETPUrunctionframeram(name)
```

Size of the PRAM required, in bytes, for the ETPU_function (or ETPU_function_group) *name* or *function number*. The value consists of the total PRAM requirements for the function parameters and the local `static` variables, the function frame. It is expressed as an integer constant.

Function frames must start on an 8-byte boundary. Therefore, this value will be rounded up to the nearest 8-byte multiple.

The following declaration will create a value for the amount of RAM needed by three functions.

```
#pragma write c, (#define REQUIRED_RAM ( ::ETPUfunctionframeram(1) \
                                + ::ETPUfunctionframeram(2) \
                                        + ::ETPUfunctionframeram(3) ));
```

The output in *basename*`_CPU.c` appears as follows:

```
#define REQUIRED_RAM ( 8 \
                       + 16 \
                       + 64 )
```

8.1.10. ::ETPUglobalimage, ::ETPUglobalimage32

```
::ETPUglobalimage

::ETPUglobalimage32
```

An image of the global PRAM variable space with initial values, expressed in either byte or 32-bit hexadecimal values.

The following declaration will emit an image of global PRAM values.

```
#pragma write c, ( const int global_var_init[] = { ::ETPUglobalimage }; );
```

The output in *basename*`_CPU.c` appears as follows:

```
const int global_var_init[] = { 1, 2, 3, 4, 5, 6, 7, 8,
                                3, 5, 7, 9, 2, 4, 6, 8,
```

and so on. The same effect can be accomplished with 32-bit values:

```
#pragma write c, ( const int32 global_var_init[] = { ::ETPUglobalimage32 }; );
```

The output in *basename*_CPU.c appears as follows:

```
const int32 global_var_init[] = { 0x12345678, 0x90123456,
                                   0x78901234, 0x56789012,
```

and so on. Either of these arrays allow the CPU program to initialize global PRAM.

8.1.11. ::ETPUglobalinit, ::ETPUglobalinit32

```
::ETPUglobalinit

::ETPUglobalinit32
```

A list of all initialized global variables. This list has the same content as ::ETPUglobalimage/::ETPUglobalimage32. Each element in the list is in turn a C macro in the form

```
__etpu_globalinit(location, value)
```

The following declaration will generate two arrays for globals initialization: one of locations, the other of values.

```
#pragma write c, (
#define __etpu_globalinit(location, value) location,

/* Initialize eTPU globals: locations */
const gl_init_locations[] = { ::ETPUglobalinit };
/* end global locations */

#define __etpu_globalinit(location, value) value,

/* Initialize eTPU globals: locations */
const gl_init_values[] = { ::ETPUglobalinit };
/* end global values */
);
```

The output in *basename*_CPU.c appears as follows:

```
#define __etpu_globalinit(location, value) location,

/* Initialize eTPU globals: locations */
const gl_init_locations[] = { __etpu_globalinit(0x0000, 0x01) };
/* end global locations */
```

```
#define __etpu_globalinit(location, value) value,

/* Initialize eTPU globals: locations */
const gl_init_values[] = { __etpu_globalinit(0x0000, 0x01) };
/* end global values */
```

Note that no extra value need be appended to each array to terminate the comma-separated list of values.

Alternatively, you can take a different approach (this example using `::ETPUglobalinit32` instead):

```
#define __etpu_globalinit(location, value) *(sharedram + location) = value;

::ETPUglobalinit32
```

Compiling the code will expand the global variable initializations into a form suitable for eTPU initialization undertaken by the host:

```
#define __etpu_globalinit(location, value) *(sharedram + location) = value;

__etpu_globalinit32(0x0000, 0x01234567)
```

which will in turn preprocess to the code

```
#define __etpu_globalinit(location, value) *(sharedram + location) = value;

*(sharedram + 0x0000) = 0x01234567;
```

8.1.12. ::ETPUglobals

```
::ETPUglobals
```

A list of all global eTPU variables exposed to host CPU code. Each element in the list is in turn a C macro in the form

```
__etpu_globals(name, datatype, address)
```

The following declaration will generate the list of globals accessible to the host:

```
#pragma write c, (
#define __etpu_globals(name, datatype, address) \
```

```
void write_##name (datatype value) { memset(address, (datatype)value,
sizeof(datatype)); } \
datatype read_##name (void) { return (datatype)memread(address, sizeof(datatype));
}

/* Initialize eTPU Globals */
::ETPUglobals
/* end globals */
);
```

The output in *basename*_CPU.c appears as follows:

```
#define __etpu_globals(name, datatype, address) \
void write_##name (datatype value) { memset(address, value, sizeof(datatype)); } \
datatype read_##name (void) { return (datatype)memread(address, sizeof(datatype));
}

/* Initialize eTPU Globals */
_etpu_globals(one, int32, 0x0000)
/* end globals */
```

Compiling the code will expand the global variable initializations into accessors and mutators suitable for use by the host program.

8.1.13. ::ETPUimagesize

```
::ETPUimagesize
```

A value representing the size of the declared eTPU SCM image (in bytes). This is useful in initially writing the executable image to SCM.

Compare with ::ETPUcodeimagesize, above.

8.1.14. ::ETPUliteral

```
::ETPUliteral
```

A literal value to pass to the host interface code unchanged.

```
::ETPUliteral start character content end character
```

where *start character* is one of (, [, {, or <, and *end character* is the corresponding),], }, or >. If you need to put one of these syntax elements, unmatched, within *content*, choose another to use as the start and end characters for ::ETPUliteral.

For instance:

```
#pragma write c, ( /* generated by ::ETPUliteral(::ETPUglobalimage) */ );
#pragma write c, ( const int global_var_init[] = { ::ETPUglobalimage }; );
```

will generate a host interface file containing:

```
/* generated by ::ETPUglobalimage */
const int global_var_init[] = { 1, 2, 3, 4, 5, 6, 7, 8,
```

and so on.

C comments within **::ETPUliteral** will not be stripped out.

8.1.15. ::ETPUmaxrom

```
::ETPUmaxrom
```

The highest offset in SCM written by the compiler during code generation.

Offsets higher than this will be written with the value of the **#pragma option fillrom** statement, if any.

8.1.16. ::ETPUmisc

```
::ETPUmisc

::ETPUmisc(size)
```

The MISC checkword for the application, expressed as an integer constant.

If given no parameter or an empty parameter, ::ETPUmisc returns the MISC value for the entire SCM image. If given an size parameter, ::ETPUmisc expands to the MISC value for the program from offset 0 of the given size in bytes, rounded up to the 32-bit word. The *size* may be a host interface macro like ::ETPUmaxrom.

The following declaration provides for the loading of the MISC checkword into the appropriate eTPU register.

```
#pragma write c, ( output(ETPUMISCCMPR, ::ETPUmisc); );
```

The output in *basename*_CPU.c appears as follows:

```
output(ETPUMISCCMPR, 0x12345678);
```

Alternatively, you can use:

```
#pragma write c, ( output(ETPUMISCCMPR, ::ETPUmisc(::ETPUmaxrom));
```

This calculates the MISC value for the generated program image.

8.1.17. ::ETPUnames

```
::ETPUnames
```

A formatted list of names of all of the eTPU functions in numerical order. Missing function numbers are null strings.

The following declaration will generate a report of the eTPU functions in the program.

```
#pragma write c, (
/* eTPU functions:
::ETPUnames
*/
);
```

The output in *basename*_CPU.c appears as follows:

```
/* eTPU functions:
"myfunction",
"",
"",
"",
...and so on
*/
```

8.1.18. ::ETPUparameterram

```
::ETPUparameterram
```

The range of parameter RAM, in bytes, available to the CPU program (available RAM, minus eTPU globals and locals). Expressed as a comma-separated range.

The following declaration will emit a routine to clear PRAM in the area used by the program.

```
#pragma write c, (
#define clear_program_PRAM( from, to ) for (int i = from; i <= to; i++)
{ clear(i); }

clear_program_PRAM(::ETPUparammeterram);
);
```

The output in *basename*_CPU.c appears as follows:

```
#define clear_program_PRAM( from, to ) for (int i = from; i <= to; i++)
{ clear(i); }

clear_program_PRAM(0x0100, 0x0200);
```

8.1.19. ::ETPUparams

```
::ETPUparams(function number)

::ETPUparams(function name)
```

A list of parameters of the ETPU_function (or ETPU_function_group) at *function name* or *function number*, giving their names, data types, and offsets. Each element in the list is in turn a C macro in the form

```
__etpu_param(name, datatype, offset)
```

The data types are expressed as standard size-specific types, described above. Your host C compiler should have support for these types.

The following declaration will write the list of parameters for an eTPU function to a host interface file:

```
#pragma write c, (
#define __etpu_param(name, datatype, offset) datatype name @ offset;

/* Parameters for myfunction() */
::ETPUparams(myfunction)
/* end myfunction() */
);
```

The output in *basename*_CPU.c appears as follows:

```
#define __etpu_param(name, datatype, offset) datatype name @ offset;

/* Parameters for myfunction() */
__etpu_param(one, int32, 0x0000)
__etpu_param(two, int24, 0x0004)
__etpu_param(three, int32, 0x0007)
/* end myfunction() */
```

Compiling the code will expand the parameter specifications into a form suitable for passing parameters to the eTPU function through host interface code.

8.1.20. ::ETPUstaticinit, ::ETPUstaticinit32

```
::ETPUstaticinit(function number)

::ETPUstaticinit(name)

::ETPUstaticinit32(function number)

::ETPUstaticinit32(name)
```

A list of static initialization information for an ETPU_function or ETPU_function_group, specified by name or function number, and generated as 8-bit or 32-bit values. Each element in the list is in turn a C macro in the form

```
__etpu_staticinit(offset, value)
```

or, for 32-bit values,

```
__etpu_staticinit32(offset, value)
```

The following code will emit the static initialization information for `myfunction()`.

```
#pragma write c, (
#define __etpu_staticinit(offset, value) output(offset, value);

/* static initialization for myfunction */
::ETPUstaticinit(myfunction)
/* end of myfunction */
);
```

The output in *basename*`_CPU.c` appears as follows:

```
#define __etpu_staticinit(offset, value) output(offset, value);

/* static initialization for myfunction */
__etpu_staticinit(0x0000,0x01)
__etpu_staticinit(0x0001,0x02)
__etpu_staticinit(0x0002,0x04)
/* end of myfunction */
```

For code that uses `::ETPUstaticinit32` instead, the macro calls will have 32-bit values:

```
#define __etpu_staticinit(offset, value) output(offset, value);

/* static initialization for myfunction */
__etpu_staticinit32(0x0000,0x01020304)
__etpu_staticinit32(0x0004,0x05060708)
__etpu_staticinit32(0x0008,0x09101112)
/* end of myfunction */
```

Compiling the code will expand the static initialization specifications into a form suitable for initializing the eTPU function through host interface code.

8.1.21. ::ETPUsymboltable

```
::ETPUsymboltable
```

The entire symbol table of the eTPU program. Each element in the list is a C macro in the form

```
__etpu_symbol(name, datatype, offset/location, PCrangeLow, PCrangeHi)
```

The following declaration will emit the symbol table for the eTPU program that is being compiled.

```
#pragma write c, (
#define __etpu_symbol(name, datatype, offset_location, PCrangeLow, PCrangeHi)

/* Symbol Table
::ETPUsymboltable
*/
);
```

The output in *basename*_CPU.c appears as follows:

```
#define __etpu_symbol(name, datatype, offset_location, PCrangeLow, PCrangeHi)

/* Symbol Table
__etpu_symbol(i                                ,sint24        , 0x0001 , 0x0224 ,
0x022B )
__etpu_symbol(j                                ,sint24        , 0x0005 , 0x0224 ,
0x022B )
__etpu_symbol(k                                ,sint32        , 0x0009 , 0x0224 ,
0x022B )
__etpu_symbol(func                             ,**            , 0x0004 , 0x022C ,
0x022F )
*/
```

In this example, the double asterisk type noted for `func()` is a special notation for functions.

Preprocessing the code will expand the symbols into a form suitable for use in documentation or in debugging applications.

8.1.22. ::ETPUlocation

```
::ETPUlocation(name)

::ETPUlocation(function name, name)
```

```
::ETPUlocation(function number, name)
```

The location in PRAM, as an offset, of the variable *name*. The macro has two forms, one for global variables, and one for **ETPU_function** parameters, and ETPU_function_group group-globals. The offset is based on the channel parameter base set by the host CPU program (in the case of function parameters) or of the PRAM itself (in the case of globals).

If you specify an **ETPU_function** name or number, *name* must be a parameter within the eTPU function. Access to static local variables is not permitted.

eTPU_C generates locations for structure members, including packed integers. Use `::ETPUtype` to learn more about the structure members and packed integers reported by `::ETPUlocation`.

8.1.23. ::ETPUtype

```
::ETPUtype(name)

::ETPUtype(function name, name)

::ETPUtype(function number, name)
```

The data type of the variable *name*. *name* may be a simple variable, a structure, array or union, or a structure or union member.

The macro has two forms: one for global variables, and one for **ETPU_function** parameters and **ETPU_function_group** group-globals. If you specify an **ETPU_function** name or number, *name* must be a parameter or local variable within the eTPU function (or group).

If *name* is a structure or union, eTPU_C will generate a series of macro calls with parameter information describing the variable. Define macros in the host interface information to expand these macro calls.

- `__etpu_struct_data(data_type)`

  This macro call is used to report structure entries of one of the eTPU_C simple types. *data_type* may be one of `uint8/16/24/32`, `sint8/16/24/32`, `struct` or `struct_array`.
  The structure may be walked by the macro calls so that only the leaf types are examined. structs and unions that are part of the structure are identified as a `struct`; this implies that the member has members itself.

- `__etpu_struct_array(array_type)`

  Arrays in a structure display this macro call, where *array_type* is one of `uint8/16/24/32`, `sint8/16/24/32`, `struct` or `struct_array`.
  Array sizing information needs to be brought out as a constant elsewhere, using `::ETPUsizeof`.

- `__etpu_struct_packed_int(sign, field_size_bits, field_offset_bits)`

  This macro call represents packed data types that are stored in structure fields. In eTPU_C, packed data types are always allocated so that no field may cross a 32 bit boundary.
  A packed integer will be indicated as `signed` or `unsigned`. The macro call includes the size in bits of the packed integer member, and the offset within the 32-bit word. The least-significant bit number is 0.

The data types are expressed as ISO standard size-specific types. Your host C compiler should have support for these types.

8.1.24. ::ETPUsizeof

```
::ETPUsizeof(name)

::ETPUsizeof(function name, name)

::ETPUsizeof(function number, name)
```

The size in bytes of the variable *name*.

The macro has two forms: one for global variables, and one for **ETPU_function** parameters and **ETPU_function_group** group-globals. If you specify an **ETPU_function** name or number, *name* must be a parameter or local variable within the eTPU function.

8.1.25. ::ETPUfunctionname, ::ETPUfunctionnumber

```
::ETPUfunctionname(function number)

::ETPUfunctionnumber(name)
```

The name or function number of an eTPU function, given the opposite data.

This function also returns the appropriate data for **ETPU_function_group**s.

8.1.26. ::ETPUengine

```
::ETPUengine(function number)

::ETPUengine(name)
```

The engine upon which the eTPU function will run. Expands to `1` or `2`.

9. Useful #pragmas

9.1. #pragma write

```
#pragma write char, (text);
```

#pragma write outputs macro-expanded *text* to a host interface file. *char* is an alphabetic character from a to z. The *char* is case-insensitive.

The output file name is the base name of the eTPU_C application with _CPU appended, and with *char* as an extension. There is no need to explicitly open or close any file: the compiler creates a host interface file at the first **write** directive.

For example, to write to a C source module, invoke:

```
#pragma write c, (// this comment appears in the host interface C file);
```

and to write to a C header file,

```
#pragma write h, (// this comment appears in the host interface header file);
```

Compiling C source modules into object files will not create host interface files. eTPU_C generates host interface files when creating the final executable at link time. The **write** directives and messages are also stored in the .COD file.

#pragma write directives write the value within parentheses with all macros expanded: other elements, including C comments, are emitted verbatim.

10. Appendix

This section includes other useful information extracted from eTPU_C documentation.

10.1. Building Software

Building Software

There are three ways to use eTPU_C to build eTPU executables. The command line method is useful when working with your preferred IDE.

1. Create a new project within BCLIDE. When you start eTPU_C, BCLIDE opens.
   1. Choose *Project*|*New Project*
   2. Fill in the Project Properties. For more information, see BCLIDE online help.

      Note that ELF does not appear as a *Hex Dump File* option. Choose the *Compiler* tab, and add `+delf` to the *Additional Options* entry to cause the compiler to emit an ELF executable.

      Click *OK* and then choose *Save Project*.
   3. Choose *File*|*New* to create C modules as usual.

      Compose your eTPU program.
   4. To compile, press F9 or choose *Compile*|*Compile*.
2. Start the `etpu_c.exe` executable. A console opens, showing important controls for eTPU_C. You can use this console to compile your main C module. Note that ELF is a *Hex Dump File* option in the eTPU_C console.
3. Invoke the eTPU_C compiler from the Windows command line. eTPU_C is a Windows executable, but it returns a valid exit status suitable for use in scripts and Makefiles. The command line is:

   ```
   etpu_c.exe options sourcefile.c
   ```

   The command line parameters are:

> *Important:*
>
> Paths with spaces (for example "`\My Documents`") need to be quoted at the command prompt. This is a requirement of your command processor or shell; see your operating system documentation for more information.

Note:

When invoking eTPU_C from a DOS-based IDE, consider using the **`start.exe`** command supplied with Windows. Use

```
start /w eTPU_C.exe
```

to invoke the compiler, wait until it completes, and return the return status of the program. The compiler returns 0 for success, 1 for failure with errors.

11. eTPU_C Glossary

The eTPU has an architecutre all its own, so it brings a lot of terminology with it.

These are some common eTPU_C terms.

alternate

A keyword used in **`#pragma ETPU_function`** statements to indicate Alternate Channel Encoding.

channel

The eTPU hardware attached to each pair of input and output pins. During ETPU_function execution, access to the channel hardware takes place through the `channel` structure.

channel conditions

The collection of channel states that invoke an eTPU function. They include timebase matches or pin transitions, flag states, and host service requests. They are specified by logical operations in the top-level `if()/else if()/else` structure of an ETPU_function.

channel structure

A C structure specially interpreted by the compiler. Using the channel structure generates eTPU subinstructions. The associated `chan` type is declared in `etpuc.h`.

common function frame

A function frame shared by all channels configured for ETPU_functions in the group.

DIOB

Data Input Output Buffer. A register used for indirect operations on PRAM. eTPU_C makes heavy use of it in generated code. You can access it directly with variables of type `register_diob`.

engine

One of two eTPU execution engines in a typical eTPU implementation. Each engine has its own

ALU, registers, and channel hardware. Two engines share code memory, parameter ram, and the interface to the host.

entry table

A table that holds entry points for all threads of all ETPU_functions. eTPU_C generates this structure automatically.

It can be relocated using the **`#pragma entryaddr`** directive.

eTPU_function

A C function specially declared as an entry point for eTPU programs.

Fractional Math

Math operations on values between 0 and 1, or -1 and 1.

On eTPU_C, `fract24` variables hold 24-bit values. Though eTPU_C does have specific fractional math capability, fractional operations may also be accomplished by regular math operations.

Function Frame

The logical allocation of PRAM for ETPU_function parameters and static variables.

Each assignment of an EPTU_function to a channel has a separate function frame.

Group

One or more global variable, C function, or ETPU_function declarations, logically related and declared to be a logical unit with a name. May be safely re-used on multiple sets of channels.

Group-global

A variable that exists outside function scope, but within the scope of a Group. Not accessible to functions outside the Group.

Host

The processor that has one or more eTPUs as peripherals.

The processor is responsible for programming the eTPU, setting up eTPU function parameters, and starting it.

Host Interface Files

Files generated by eTPU_C for the use of the compiler creating the host program. eTPU_C can generate useful information with Host Interface Macros.

Host Interface Macros

Built-in macros expanded into information about the program by eTPU_C. eTPU_C can generate Host Interface Files to communicate this information to the host.

Host Service Request

An interrupt issued by the host processor.

To respond to a host service request, the ETPU_function must test the `hsr` variable in the uppermost **`if()/else if()/else`** structure. Omitting the test is equivalent to testing against 0, the do-nothing setting for the host-side request register.

if()/else if()/else structure

Specially-optimized code in an ETPU_function.

This uppermost code in an ETPU_function is a series of tests of the condition codes. This code never results in generated instructions: the compiler populates the entry table based on the tests.

MISC

The Multiple Input Signature Calculator.

eTPU_C calculates the appropriate signature value and makes it available to host interface files with the `::ETPUmisc` host interface macro.

Parameters

Parameters are variables holding information passed from host to eTPU functions, or function to subordinate function.

ETPU_function parameters are declared exactly as other C function parameters. ETPU_function parameters are configured by the host processor during initialization, and remain in force until the

host changes the channel's configuration.

PRAM

Parameter RAM. The RAM shared by one or more eTPU engines and the host. eTPU_C generates host interface information for this RAM. The host program allocates RAM space for the ETPU_function parameters and statics during ETPU_function and channel initialization.

Previously described as SPRAM (Shared Parameter RAM).

preloaded parameters

Parameters that are loaded into the P and DIOB registers at the beginning of thread execution. eTPU will preload either parameters 0 and 1, or 2 and 3, depending upon the setting in the entry table.

The compiler selects which parameters to preload, based on generated code. The setting appears in the listing file, in the entry:

```
00C4 40 80        03 S02 P01 ME 0200  HSR 1     lsr x ...
```

The setting appears in the sixth field: **`P01`** for preload 0 and 1, **`P23`** for preload 2 and 3.

post processing macros

An alternative name for host interface macros.

standard

A keyword used in **`#pragma ETPU_function`** statements to indicate Standard Channel Encoding.

standard header files

The files commonly **`#include`**d into eTPU_C program modules.

- `etpuc.h` defines channel structure, registers and processors specific constants.
- `etpuc_common.h` assigns common names and operation definitions using the eTPU channel instructions.
- `etpuc_util.h` eTPU application utilities including angle math.

thread

An entry point and code run in response to certain channel conditions. Effectively, the body of one branch of the top-level `if()/else if()/else` structure of an ETPU_function.

Index