1. "#pragma write" Host Interface Support

Although the ETEC auto-generation of host interface information (defines file, idata files, scm files) should provide all the necessary information for the host CPU to initialize and interface with the eTPU, ETEC also supports the "#pragma write" capability that has been used to generate this information in existing code. By default, #pragma write support is enabled in the ETEC linker; it can be disabled by specifying the "-pw-" option to the linker. Final #pragma write processing can only occur in the linker because many #pragma write outputs require fully resolved executable data.

The #pragma write capability allows greater user control in what host interface data is exported, how it is arranged, formatted, etc. This #pragma has the format

```
#pragma write letter, (text)
```

Where ***letter*** is an alphabetic character, a-z or A-Z. It is not case-sensitive and thus b is equivalent to B, etc. For each different letter used, a file is generated. By default, the file has the name ***<link output filename>_CPU.letter***. At the first instance of a particular letter, the file is opened for writing, erasing any previous file with the same name. The ***text***, after processing, is output to the file associated with the specified ***letter***. A new-line character is appended to the text automatically. Any following #pragma write commands the same letter append their text to the previously opened file.

The processing of the ***text*** is as follows:

- Comment begin tokens // and /* are ignored by the C preprocessor – thus they pass to the host interface output file unchanged.
- Macro replacement is done, except inside the ::ETPUliteral host interface macro (described in more detail below). This macro replacement is done even within /* or */ or after // tokens.
- Any special ::ETPU host interface macros are processed and replaced.

The #pragma write capability works in conjunction with a set of built-in ::ETPU host interface macros. These macros provide access and output capability for compiler/linker generated data. Ensuing sections provide details on these host interface macros.

Below are several examples of the #pragma write capability. The presence of the following text in the eTPU source code PWM.c:

```
#pragma write h, (/*************************************************************);
#pragma write h, ( * WARNING this file is automatically generated DO NOT EDIT IT! *);
#pragma write h, ( * This file provides an interface between eTPU code and CPU    *);
#pragma write h, ( * code. All references to the PWM function should be made with *);
#pragma write h, ( * information in this file. This allows only symbolic         *);
#pragma write h, ( * information to be referenced which allows the eTPU code to be*);
#pragma write h, ( * optimized without effecting the CPU code.                   *);
#pragma write h, ( *************************************************************/);
#pragma write h, (/* Function Configuration Information */);
#pragma write h, (::ETPUliteral(#define PWM_FUNC_NUMBER) ::ETPUfunctionnumber(PWM) );
#pragma write h, (::ETPUliteral(#define PWM_TABLE_SELECT) ::ETPUentrytype(PWM) );
#pragma write h, (::ETPUliteral(#define PWM_NUM_PARMS) ::ETPUram(PWM) );
#pragma write h, ( );
#pragma write h, (/* Host Service Request Definitions */);
#pragma write h, (::ETPUliteral(#define PWM_INIT) PWM_INIT );
#pragma write h, (::ETPUliteral(#define PWM_IMM_UPDATE) PWM_IMMED_UPDATE );
#pragma write h, (::ETPUliteral(#define PWM_CO_UPDATE) PWM_COHERENT_UPDATE );
```

Results in the following output to a file named PWM_CPU.h (output of ::ETPU macros may vary depending upon compile/link, etc.):

```
/************************************************************
 * WARNING this file is automatically generated DO NOT EDIT IT! *
 * This file provides an interface between eTPU code and CPU    *
 * code. All references to the PWM function should be made with *
 * information in this file. This allows only symbolic          *
 * information to be referenced which allows the eTPU code to be*
 * optimized without effecting the CPU code.                    *
 ************************************************************/
/* Function Configuration Information */
#define PWM_FUNC_NUMBER  0
#define PWM_TABLE_SELECT 1
#define PWM_NUM_PARMS 0x0018

/* Host Service Request Definitions */
#define PWM_INIT 7
#define PWM_IMM_UPDATE 3
#define PWM_CO_UPDATE 5
```

1.1. Supported ::ETPU Host Interface Macros

All the #pragma write built-in host interface macros begin with "::ETPU". Some are function-like macros that take parameters, while others are object-like macros. The function-like macros can have white space between the macro name and the opening parentheses. Some ::ETPU host interface macros themselves expand to function-like macros. These output macros are not defined by the #pragma write output in any automated way – it is up to the user to define them in an appropriate manner. The ::ETPU host interface macros are only applicable inside of #pragma write constructs.

Several host interface macros output type information. The table below describes the type name output for each supported C type.

ETEC (C99) C type name	Host Interface Macro output type name	Description
_Bool	bool	C99 _Bool type (0 or 1). May use 1 bit or 8 bits for storage depending upon compiler options, scope, etc.
signed char int8	sint8	signed 8-bit integer
unsigned char unsigned int8	uint8	unsigned 8-bit integer
short, signed short int16	sint16	signed 16-bit integer
unsigned short unsigned int16	uint16	unsigned 16-bit integer

int, signed int int24	sint24	signed 24-bit integer
unsigned int unsigned int24	uint24	unsigned 24-bit integer
long int, signed long int int32	sint32	signed 32-bit integer
unsigned long int unsigned int32	uint32	unsigned 32-bit integer
fract8, signed fract8	sfract8	signed s.7 fixed point type
unsigned fract8	ufract8	unsigned 0.8 fixed point type
fract16, signed fract16 short _Fract, signed short _Fract	sfract16	signed s.15 fixed point type
unsigned fract8 unsigned short _Fract	ufract16	unsigned 0.16 fixed point type
fract24, signed fract24 _Fract, signed _Fract	sfract24	signed s.23 fixed point type
unsigned fract24 unsigned _Fract	ufract24	unsigned 0.24 fixed point type
<type> * (pointer)	pointer	C pointer (to any type)
<type> A[N]	array	C array (of any type)
struct	struct	C structure
union	union	C union
function	function	C function
label	label	C label
_Bool bit array	bit_array	ETEC _Bool bit array (array of _Bool type packed into bits)
void	void	void type

For function-like host interface macros, if the provided parameter(s) is not valid, an error message is injected into the host interface output file. Such errors do not generate linker errors. Where “func name” or “func number” are used, these refer to eTPU function names or their function number – see the #pragma ETPU_function description for further information. Note that these host interface macros also work with ETEC enhanced mode.

When referencing a class (e.g. via ::ETPUfunctionframeram) only the class name need be used, but when outputting a function number such as with ::ETPUfunctionnumber the class and entry table should be referenced using “Class::Table” syntax.

The following provides details on supported ::ETPU host interface macros:

1.1.1. ::ETPUautodefinename(<global var ref>), ::ETPUautodefinename(<func number> or <func name>, <channel frame var ref>)

This host interface macro resolves to the macro name that is output in the ETEC auto defines header file for the specified variable. Either global variables or channel frame variables can be referenced. <func name> is the name of an eTPU function, or in ETEC enhanced mode, an ETEC class.

```
int24 g_s24;
#pragma write h, (::ETPUliteral(const UINT32 ETPU_G_S24_OFFSET ) =
::ETPUautodefinename( g_s24 ); );
```

becomes

```
const UINT32 ETPU_G_S24_OFFSET = _GLOB_VAR24_g_s24_;
```

Note that members of struct/union-type variables can be specified as well – doing so provides the macro representing the offset of the specified member within its containing struct.

```
struct S1 { int a, int b; };
struct S2 { struct S1 s1; char x, y; };
struct S2 g_s2;
#pragma write h, (::ETPUliteral(const UINT32 ETPU_G_S2_X_OFFSET = )
::ETPUautodefinename(g_s2.x);  );
```

yields

```
const UINT32 ETPU_G_S2_X_OFFSET = _GLOB_MEMBER_BYTEOFFSET_S2_x_;
```

To build the total offset to a variable member, the following could be written

```
#pragma write h, (::ETPUliteral(const UINT32 ETPU_G_S2_S1_b_OFFSET = )
(::ETPUautodefinename(g_s2) + (::ETPUautodefinename(g_s2.s1) +
(::ETPUautodefinename(g_s2.s1.b));  );
```

1.1.2. ::ETPUchanframebase(<eTPU engine count>)

This macro outputs the auto-calculated recommended value for the start of channel frame allocation in data memory. It takes into account global data, as well as engine data and stack usage, if any. Since the value can depend upon whether the configuration is single or dual eTPU engine, the engine count must be provided. The output is the equivalent of auto-defines macro _CHANNEL_FRAME_1ETPU_BASE_ADDR or _CHANNEL_FRAME_2ETPU_BASE_ADDR.

1.1.3. ::ETPUcode, ::ETPUcode32

These macros output the code image as a list of comma separated hexadecimal data. The entire code image, as specified via the –codesize linker option, is output. ::ETPUcode outputs the data in 8-bit values, while ::ETPUcode32 outputs the data in 32-bit values.

```
#pragma write h, (const uint32_t etpu_code[] = { ::ETPUcode32 }; );
#pragma write h, (const uint8_t etpu_code8[] = { ::ETPUcode }; );
```

Generates

```
const uint32_t etpu_code[] = { 0x42014201,0x42014201,0x42010200,0x42014201,
                               0x42014201,0x42014201,0x42014201,0x42014201,
...
                               0xFFD04107,0xFFD04107,0xFFD04107,0xFFD04107
                                };
const uint8_t etpu_code8[] = { 0x42,0x01,0x42,0x01,0x42,0x01,0x42,0x01,
                               0x42,0x01,0x02,0x00,0x42,0x01,0x42,0x01,
...
                               0xFF,0xD0,0x41,0x07,0xFF,0xD0,0x41,0x07
                                };
```

1.1.4. ::ETPUcodeimagesize

The ::ETPUcodeimagesize macro is defined to the code image size, in bytes.

```
#pragma write h, (const uint32_t etpu_code_size = ::ETPUcodeimagesize; );
```

Outputs

```
const uint32_t etpu_code_size = 0x1800;
```

1.1.5. ::ETPUenginebase_a(<eTPU engine count>), ::ETPUenginebase_b(<eTPU engine count>)

This macro outputs the auto-calculated recommended value for the engine-relative base address(es) in data memory. It takes into account global data, and stack usage, if any. Since the value can depend upon whether the configuration is single or dual eTPU engine (at least in the case of engine A), the engine count must be provided. The output is the equivalent of auto-defines macro
_ETPU_A_ENGINE_1ETPU_RELATIVE_BASE_ADDR,
_ETPU_A_ENGINE_2ETPU_RELATIVE_BASE_ADDR, or
_ETPU_B_ENGINE_2ETPU_RELATIVE_BASE_ADDR.

[NOTE: available in release 2.62A and newer.]

1.1.6. ::ETPUenginedatasize

Defined to be the size of allocated engine address space data, including any engine scratchpad data, rounded up to the nearest 4-byte boundary. Equivalent to the auto-defines macro _ENGINE_DATA_SIZE_. "::ETPUenginememsize" acts as a synonym.

1.1.7. ::ETPUengineinit, ::ETPUengineinit32

These macros export engine-relative variable initial data information in the form of macros with address and data information. ::ETPUengineinit outputs the data in 8-bit chunks, while ::ETPUengineinit32 outputs it in 32-bit chunks. The amount of data output covers the entire engine-relative variable allocation, regardless of whether initial values are non-zero or not. The output macros have the form

__etpu_engineinit(<byte address>, <value>)

or

__etpu_engineinit32(<byte address>, <value>)

::ETPUengineinit outputs a macro per byte, while ::ETPUengineinit32 outputs a macro per 32-bit word.

```
_ENGINE int8 g_s8 = 0xab;
_ENGINE int24 g_s24 = 0x101010;
#pragma write a, (::ETPUengineinit);
#pragma write a, (::ETPUengineinit32);
```

Generates

```
__etpu_engineinit(0x0000,0xAB)
__etpu_engineinit(0x0001,0x10)
__etpu_engineinit(0x0002,0x10)
__etpu_engineinit(0x0003,0x10)

__etpu_engineinit32(0x0000,0xAB101010)
```

Note that if the scratchpad is located in engine-relative space, such variables are also included in the initialization data, even though any initialization of such is typically overwritten during run-time execution.

1.1.8. ::ETPUentrybase

Defined to the address of the base of the entry table.

1.1.9. ::ETPUentrytype(<func number> of <func name>)

This macro resolves to the encoding format of the specified eTPU function (name or number). The format can be either standard (0) or alternate (1).

```
// PWM is an eTPU-C mode eTPU function of alternate entry type
#pragma write h, (::ETPUliteral(#define PWM_TAB_SELECT) ::ETPUentrytype(PWM)
);
// Test1 is a standard entry table in class Test
#pragma write h, (::ETPUliteral(#define TEST1_TAB_SELECT)
::ETPUentrytype(Test::Test1) );
```

Generates

```
#define PWM_TAB_SELECT 1
#define TEST1_TAB_SELECT 0
```

1.1.10. ::ETPUfilename(<filename (and path)>)

This macro has the side effect of associating the specified file (may include relative or absolute path) with the #pragma write letter, rather than the default file name of ***<base name>_CPU.letter***. If a file was previously associated with the letter it is closed. The macro itself expands to nothing.

```
/* Information exported to Host CPU program etpu_pwm_auto.h */
/* located in subdirectory 'cpu' */
#pragma write h, (::ETPUfilename (cpu/etpu_pwm_auto.h));
```

1.1.11. ::ETPUfunctionframeram(<func number> or <func name>)

This is a function-like macro that either takes a function number (0 through 31), or a function name (or ETEC class name). The replacement text of the macro is the size of the specified function's channel frame (already rounded up to the next largest 8-byte boundary).

```
#pragma write h, (::ETPUliteral(#define PWM_NUM_PARMS) ::ETPUfunctionframeram(PWM) );
```

Results in

```
#define PWM_NUM_PARMS 0x0018
```

1.1.12. ::ETPUfunctionname(<func number>)

This macro resolves to the function name of the eTPU function (or class) corresponding to the specified function number, with a space on either side.

1.1.13. ::ETPUfunctionnumber(<func name>)

::ETPUfunctionnumber is defined to the function number of the specified eTPU function (or Class::Table).

```
#pragma write h, (::ETPUliteral(#define FUNC_NUMBER)
::ETPUfunctionnumber(PWM) );
#pragma write h, (::ETPUliteral(#define TEST1_FUNC_NUMBER) \
::ETPUfunctionnumber(Test::Test1) );
```

Outputs

```
#define PWM_FUNC_NUMBER  0
#define TEST1_FUNC_NUMBER  1
```

1.1.14. ::ETPUfunctionstackbase(<func number> or <func name>)

This macro outputs the channel fame offset for the stack base. If the function does not use the stack an invalid offset value of 0xFFFF is output. For eTPU functions that utilize the stack, the 24-bit memory slot at the offset provided by this macro must be initialized with the value of the stack base for the given eTPU engine (see ::ETPUstackbase_a/b). The output is equivalent to the auto-defines macro _CPBA24_<func name>___STACKBASE_.

[NOTE: available in release 2.62B and newer.]

1.1.15. ::ETPUglobaldatasize

Defined to be the size of allocated global address space data, including any global scratchpad data, rounded up to the nearest 4-byte boundary. Equivalent to the auto-defines macro _GLOBAL_DATA_SIZE_.

1.1.16. ::ETPUglobalimage, ::ETPUglobalimage32

These macros export global variable initial data information in the form of comma-separated hexadecimal values. Typically they are used to output array initializers. ::ETPUglobalimage outputs the data in 8-bit chunks, while ::ETPUglobalimage32 outputs

it in 32-bit chunks. The amount of data output covers the entire global variable allocation, regardless of whether initial values are non-zero or not.

```
int8 g_s8 = 0xab;
int24 g_s24 = 0x101010;
#pragma write h, (const uint32_t etpu_globals[] = { ::ETPUglobalimage32 };
);
#pragma write h, (const uint8_t etpu_globals8[] = { ::ETPUglobalimage }; );
```

Yields

```
const uint32_t etpu_globals[] = { 0xAB101010,0x00000000
                                  };
const uint8_t etpu_globals8[] = { 0xAB,0x10,0x10,0x10,0x00,0x00,0x00,0x00
                                  };
```

1.1.17. ::ETPUglobalinit, ::ETPUglobalinit32

These macros output the same data as ::ETPUglobalimage & ::ETPUglobalimage32, except they do so in the form of a series of macros. The output macros have the form

__etpu_globalinit(<byte address>, <value>)

__etpu_globalinit32(<byte address>, <value>)

::ETPUglobalinit outputs a macro per byte, while ::ETPUglobalinit32 outputs a macro per 32-bit word.

```
int8 g_s8 = 0xab;
int24 g_s24 = 0x101010;
#pragma write a, (::ETPUglobalinit);
#pragma write a, (::ETPUglobalinit32);
```

Generates

```
__etpu_globalinit(0x0000,0xAB)
__etpu_globalinit(0x0001,0x10)
__etpu_globalinit(0x0002,0x10)
__etpu_globalinit(0x0003,0x10)
__etpu_globalinit(0x0004,0x00)
__etpu_globalinit(0x0005,0x00)
__etpu_globalinit(0x0006,0x00)
__etpu_globalinit(0x0007,0x00)

__etpu_globalinit32(0x0000,0xAB101010)
__etpu_globalinit32(0x0004,0x00000000)
```

1.1.18. ::ETPUglobals

Exports a list of all global variables in the form of a macro that includes the name, type and address. The type is one of the type names listed in section 1.1. Global variables such as:

```
int8 g_s8 = 0xab;
int24 g_s24 = 0x101010;
```

export as

```
__etpu_globals(g_s8                              ,sint8     ,0x0000)
__etpu_globals(g_s24                             ,sint24    ,0x0001)
```

1.1.19. ::ETPUliteral ( ), { }, [ ], < >

The ::ETPUliteral host interface macro bounds a set of text that is to be passed to the output file exactly as-is. The text can be bound by one of 4 sets of tokens, thus allowing any of the other 3 to be in the literal text unmatched. The main use of ::ETPU literal is to with output unmatched parentheses, or to ensure that no macro replacement is done on the text.

1.1.20. ::ETPUlocation(<global var ref>), ::ETPUlocation(<func number> or <func name>, <channel frame var ref>)

The ::ETPUlocation host interface macro provides location information of the specified global or channel frame variable. For channel frame variables, the output address is a channel-relative address. Simple channel variable locations can be exported as follows:

```
#pragma write h, (::ETPUliteral(#define PWM_PERIOD_OFFSET) ::ETPUlocation
(PWM, Period) );
#pragma write h, (::ETPUliteral(#define PWM_ACTIVE_OFFSET) ::ETPUlocation
(PWM, ActiveTime) );
```

Resulting in

```
#define PWM_PERIOD_OFFSET  0x0001
#define PWM_ACTIVE_OFFSET  0x0005
```

Host-side code can use this information to access interface data shared between the host and eTPU. Global variable location information is similar:

```
#pragma write h, (::ETPUliteral(#define G_S8_OFFSET) ::ETPUlocation (g_s8)
);
#pragma write h, (::ETPUliteral(#define G_S24_OFFSET) ::ETPUlocation (g_s24)
);
```

Yields

```
#define G_S8_OFFSET  0x0000
#define G_S24_OFFSET  0x0001
```

Location information of structure members can also be output by requesting location information for <struct var name>.<member name>:

```
#pragma write h, (::ETPUliteral(#define FS_ETPU_PPA_ACCUM_OFFSET)
::ETPUlocation (PPA, this.Accum) );
```

Outputs

```
#define FS_ETPU_PPA_ACCUM_OFFSET  0x0001
```

When a struct member is referenced, the output value is no the offset within the struct, but rather the member offset ***plus*** the structure location, i.e. the absolute address (or channel-relative) of the member data.

1.1.21. ::ETPUmaxrom

The ::ETPUmaxrom host interface macro resolves to the highest Shared Code Memory byte address used by the output code image. It could be entry table or a code opcode depending upon the memory map.

1.1.22. ::ETPUmisc

::ETPUmisc equates to the MISC signature for the code image, in the form of a 32-bit hexadecimal word.

```
#pragma write h, (#define FS_ETPU_MISC ::ETPUmisc);
```

Outputs

```
#define FS_ETPU_MISC 0x9A0AA5B4
```

1.1.23. ::ETPUnames

The ::ETPUnames macro resolves to a comma-separated list of the eTPU function names, ordered by function number. Function numbers with no associated code come out as empty strings.

1.1.24. ::ETPUparams(<func number> or <func name>)

This host interface macro outputs the channel frame variables of the specified eTPU function (or ETEC class). The output format is very similar that of ::ETPUglobals:

__etpu_param(<name>, <type>, <channel-relative byte address>)

1.1.25. ::ETPUram(<func number> or <func name>)

::ETPUram behaves the same as ::ETPUfunctionframeram.

1.1.26. ::ETPUsizeof(<global var ref> or <global type ref>), ::ETPUsizeof(<func number> or <func name>, <channel frame var ref>)

The ::ETPUsizeof macro resolves to the size in bytes of the object or type referenced. Global variables (or type names, e.g. the name of a structure type) can be referenced in a single argument. Channel frame variables are referenced by a function number or function/class name and the object name.

Members of variables of struct or union type can be referenced via the '.' notation. Code such as:

```
typedef struct
{
   int24 a, b;
} S1;
typedef struct
{
   S1 x;
   int8 y;
} S2;
S2 g_s2;
// _s2 is a channel var in PWM function, of type S2
#pragma write h, (// sizeof(_s2.x) == ::ETPUsizeof(PWM, _s2.x) );
#pragma write h, (// sizeof(g_s2) == ::ETPUsizeof(g_s2) );
#pragma write h, (// sizeof(g_s2.y) == ::ETPUsizeof(g_s2.y) );
#pragma write h, (// sizeof(S1) == ::ETPUsizeof(S1) );
```

would yield something like the below in the output file:

```
// sizeof(_s2.x) ==  8
// sizeof(g_s2) ==  8
// sizeof(g_s2.y) ==  1
// sizeof(S1) ==  8
```

1.1.27. ::ETPUstackbase_a(<eTPU engine count>), ::ETPUstackbase_b(<eTPU engine count>)

This macro outputs the auto-calculated recommended value for the stack base address(es) in data memory. It takes into account global data usage. Although currently the calculated value does not depend upon whether the configuration is single or dual eTPU engine, the engine count must be provided (there could be future changes in how stack base is calculated). The output is the equivalent of auto-defines macro _ETPU_A_STACK_BASE_ADDR or _ETPU_B_STACK_BASE_ADDR.

[NOTE: available in release 2.62A and newer.]

1.1.28. ::ETPUstacksize

This macro equates to the required stack size for the compiled executable. This value will be 0 if a scratchpad programming model is used, or if none of the code requires stack. It is equivalent to the auto-defined macro _STACK_SIZE_.

[NOTE: available in release 2.62B and newer.]

1.1.29. ::ETPUstaticinit(<func number> or <func name>), ::ETPUstaticinit32(<func number> or <func name>)

These macros output channel frame initialization data as a series of macros. These output macros have the form

__etpu_staticinit(<channel-relative byte address>, <8-bit data>)

__etpu_staticinit32(<channel-relative byte address>, <32-bit data>)

```
static int24 x = 0x123456; // in scope of eTPU function
...
#pragma write b, (::ETPUstaticinit(PWM));
#pragma write b, (::ETPUstaticinit32(PWM));
```

would result in

```
...
__etpu_staticinit(0x0010,0x00)
__etpu_staticinit(0x0011,0x12)
__etpu_staticinit(0x0012,0x34)
__etpu_staticinit(0x0013,0x56)
...
__etpu_staticinit32(0x0010,0x00123456)
```

Note that initial values for the full channel frame are output when these macros are used, even if the variables do not have initializers (default is 0).

1.2. Unsupported ::ETPU Host Interface Macros

The special environment extension ‘e’ processing is not supported in that no additional data is written to the file outside of that specified in the ***#pragma write text***.

The following ::ETPU host interface macros are not supported currently:

```
::ETPUcode(<size>)
::ETPUcode32(<size>)
::ETPUentry
::ETPUentrytables
::ETPUfunction
::ETPUimagesize
::ETPUmisc(<size>)
::ETPUparameterram
::ETPUsymboltable
::ETPUtype
::ETPUengine
```