Porting of eTPU C Code from Legacy Code Generation Tools to ETEC

Overview

Porting source code and build processes from other eTPU tool chains to ASH WARE's ETEC Compiler tool suite is a fairly straightforward process. However, embedded software tool chain conversions are almost never just "drop-in", and that is the case here. It is assumed the reader is familiar with programming the eTPU. This document provides a step-by-step list of the changes that must be made to port existing eTPU code to build & run using the ETEC Compiler. In general, code can be modified so that it builds as-is under both tools. There are three main areas to address: changes to the build process, changes to the eTPU source code itself, and modifications to the host / simulator interfaces. The sections below discuss each area in detail.

References

ETEC Reference Manual
ISO/IEC 9899:1999 – C Language Specification (C99)
ISO/IEC TR18037 – C Language : Extensions for Embedded Processors

Build Process

The ETEC tool suite consists of four executables, all of which operate as command line tools:

ETEC_cpp.exe	- C Preprocessor (typically not run directly by users)
ETEC_cc.exe	- eTPU/eTPU2 C Compiler
ETEC_asm.exe	- eTPU/eTPU2 Assembler
ETEC_link.exe	- eTPU/eTPU2 Linker

Both the Compiler and Assembler emit ".eao" files (eao stands for "ETEC annotated object") which currently is a text-based object file format. The Linker reads one or more .eao files, performs optimization, linking, locating tasks, and outputs an executable image. The default output format is the standard format ELF/DWARF2.0 with eTPU-specific extensions, but additional outputs are also auto-generated for use by host-code eTPU initialization code. These additional files will be discussed in detail in later sections, but for overview purposes:

<output file name base>_defines.h	- macros for function numbers, data offsets, etc.
<output file name base>_idata.c	- initialized arrays for global & channel frame data
<output file name base>_idata.h	- initialized data macros for global / frame data
<output file name base>_scm.c	- initialized array of the code image
<output file name base>_scm.h	- raw code image data
<output file name base>_struct.h	- auto-struct host-eTPU memory overlays

Since the ETEC tools are command-line based, they typically can be integrated into the user's IDE of choice, such as Eclipse, CodeWright or Developer Studio through makefile changes or custom build settings. Improved Eclipse integration (e.g. ability to click on build error and directly go to proper source line) is available by using the option "-msgstyle=GNU".

Converting an existing eTPU build process to use ETEC is generally an easy procedure when either a command line batch script or a makefile approach is used. Those who use an IDE-based build process will have to create build scripts or makefiles to best use ETEC. At ASH WARE we frequently use command line batch scripts; below is an example of a batch script that builds the Freescale PWM function with another eTPU compilation tool and ETEC.

```
echo Building Freescale PWM with an eTPU compilation tool
etpu_c  etpuc_pwm.c  +l +e +q
if  %ERRORLEVEL% NEQ 0 ( goto errors )

echo Building Freescale PWM with ETEC / ASH WARE tools
ETEC_cc.exe etpuc_pwm.c -globalscratchpad
if  %ERRORLEVEL% NEQ 0 ( goto errors )
rem use -lst to generate list files
rem use -codeSize=0x3000 to set 12K SCM
ETEC_link.exe etpuc_pwm.eao -lst -codeSize=0x3000
if  %ERRORLEVEL% NEQ 0 ( goto errors )

goto end
:errors
echo ********************************
echo        YIKES, WE GOT ERRORS!!
echo ********************************
:end
```

The above is a simple case of the compilation & link of a single source file; note that ETEC still requires individual compile & link stages in this case.

Makefiles are a nicer way to go, but dependency checking is less important with eTPU builds versus larger host-side builds since they are relatively very quick (full re-build not that expensive in terms of time). Below is part of a GNU makefile:

```
ETPUCC = ETEC_cc.exe
# disable warning 110 (unknown/unsupported #pragma)
ETPUCCFLAGS = -warnDis=110 -globalscratchpad
ETPULINK = ETEC_link.exe
ETPULINKFLAGS = -codeSize=0x1800

ETPUFUNCS = etpuc_pwm.c etpuc_ppa.c etpuc_uart.c

#
#ETPU targets
```

```
#

etpuc_set1.elf : etpuc_set1.eao
   $(ETPULINK) $< $(ETPULINKFLAGS)

etpuc_set1.eao : etpuc_set1.c $(ETPUFUNCS) makefile
   $(DEL) etpuc_set1.eao
   $(ETPUCC) $< $(ETPUCCFLAGS)
```

Note how the makefile example really builds just a single .c file into the executable – etpuc_set1.c is the "master" source file that includes all others. For good or for bad, many existing eTPU software projects have been set up this way. With ETEC and its truly distinct compilation and linking stages it is preferred to compile each source file separately into an .eao file, then link them all together. However, in order to minimize changes to legacy software ETEC supports either technique. A snippet from a makefile that utilizes the preferred technique might look like:

```
OBJECTS=etpuc_pwm.eao etpuc_ppa.eao etpuc_uart.eao

%.eao: %.c    $<
   $(CC) $(CFLAGS)  $(CINCLUDES)  $(LINCLUDES) $(PINCLUDES) $<

%.eao: %.sta $<
   $(ASM) $(AFLAGS) $<

all: $(OBJECTS)
   $(ECHO) 'Compile completed'
   $(ECHO) 'ETEC C Linker building target'
   $(LINK) $(LINK_FLAGS) $(OBJECTS) -out=etpuc_set1.elf
   $(ECHO) 'Generating sources and headers for target'
   $(ECHO) 'Target build completed.'
```

Programming Model

The ETEC compiler offers several different programming model options – stack, global scratchpad, and engine scratchpad. These options factor into how the call stack is managed, and how local variables that overflow register usage are dealt with. The stack model (default) is the most generic and works similar to most modern compilers, using pre-allocated stack space in the data memory to hold call frames and local variables. However, it is also the least efficient in terms of execution speed and code size (the eTPU instruction set has poor stack support, unfortunately).

The scratchpad models place overflow and call frame data in global (or engine memory – eTPU2 only) memory locations. The ETEC –globalscratchpad compiler option should be used to get the global scratchpad programming model, which is the default programming model available with most other tools. Programming models can be mixed across different eTPU functions in the same executable, although it is probably best to avoid such unless there is good reason.

Please see the compiler reference manual for more details on the programming models.

Source Code Conversion

Although the section title includes the term conversion, the thrust of the information discusses making source code portable/compatible with both tool sets. To this end, a key ETEC feature is that the compiler provides the predefined macro __ETEC__. Use of `#ifdef __ETEC__` as a compile switch is the key to writing portable code that can be compiled as-is by multiple tools.

Include File Changes

The first step in a conversion is to change the standard include files. The include files such as `etpuc.h` and `etpuc_common.h` that are distributed with other compilation systems are not included with ETEC. ETEC comes with its own set of three standard header files:

etpu_hw.h - defines the hardware programming model
etpu_lib.h - declarations/prototypes of ETEC library calls
etpu_std.h - standard wrappers around the low-level programming model (includes both etpu_hw.h and etpu_lib.h)

In general, it is recommended that users just include etpu_std.h and be done with it. However, in some cases users may have created their own macro header file to replace etpuc_common.h, in which case that is probably useable (assuming it includes etpu_hw.h). Great care must be taken in this case though since the user defined macros will not include ETEC constructs that ensure proper compilation and operation. An example might be

```
#define SetupMatchA(reference) { erta = reference; _AtomicBegin();
EnableMatchA(); ClearMatchALatch(); _AtomicEnd(); }
```

The EnableMatchA(); and ClearMatchALatch(); are placed within an atomic region, which means the compiler & optimizer must always keep them together in a single opcode. Other sub-instructions may join them, but they will always stay together and execute atomically.

So, if at the top of an existing source file there is code like:

```
#include <etpuc.h>          /*Defines eTPU hardware*/

#ifndef __ETPUC_COMMON_H
#include <eTPUc_common.h>   /*Standard way to use eTPU*/
#endif
```

It should be modified to look like:

```
#ifdef __ETEC__
#include <ETpu_Std.h>       /* Standard ETEC Definitions */
#else
#include <etpuc.h>          /*Defines eTPU hardware (non-ETEC) */

#ifndef __ETPUC_COMMON_H
#include <eTPUc_common.h>   /*Standard way to use eTPU*/
```



```
#endif
#endif
```

Note that since ETEC is a Windows-based toolset, file names are not case-sensitive.

Unsupported Constructs

There are several implementation-defined constructs, or non-C99 constructs, used in software developed for other eTPU compilers that are not supported by ETEC. One of these is the '@' locating/aliasing construct. This '@' is used to explicitly locate items in memory. At this time ETEC does not have a mechanism to give users explicit locating control (not really needed on the eTPU, or other techniques can be used to get the proper results – contact us if you have a case where this is an issue). The aliasing aspect of the '@' construct can actually be handled with compliant C99/TR18037 compliant constructs. In eTPU code the '@' symbol is most frequently used to overlay a structure (bit-field) on the TPR register. This is provided in ETEC in the etpu_hw.h files as follows (spec. compliant code):

```
struct tpr_struct {
   unsigned int16 TICKS   : 10;
   unsigned int16 TPR10   : 1;
   unsigned int16 HOLD    : 1;
   unsigned int16 IPH     : 1;
   unsigned int16 MISSCNT : 2;
   unsigned int16 LAST    : 1;
} register _TPR tpr_reg;
```

The symbol 'tpr_reg' with type 'struct tpr_srtruct' is aliased to register _TPR.

Syntax Compatibility Issues

ETEC is designed to be C99 and TR18037 compliant, and does thorough syntax checking per these specifications. In porting some existing code (e.g. Freescale Set 1 functions), a number of minor compatibility issues have been discovered wherein non-compliant syntax that compiles with other tools will not compile with ETEC. Generally these are very easy to fix, and they also do not generally require use if #ifdef __ETEC__ since the necessary changes will also work with other compilers. The list below covers some such issues detected, but is not exhaustive:

- `NOP;` must be `NOP();`
- Missing semicolons are sometimes allowed by other compilation tools but will induce ETEC syntax errors.
- Function parameters sharing a single declaration_specifiers will not compile. E.g. `int MyFunc(int24 width, highTime)` is required to be `int MyFunc(int24 width, int24 highTime).`
- */ followed by another */ will trigger an error as the first */ encountered marks the end of a /* … */ comment.
- Extern declarations cannot include an initializer. E.g. `extern int g_CamState = 5;` will fail to compile.

- A declaration or definition is supposed to require at least one type-specifier rather than default to “int” if there is none. ETEC makes one exception to this in that it allows an un-typed function definition which defaults to a return type of “int”.
- A label must be followed by a statement (not a declaration or end of compound statement), even if it is just a ‘;’.
- Tighter type checking in some cases will cause ETEC to issue errors/warnings for code that compiles without error/warning with other tools.

C Language Functional Compatibility Issues

This section covers issues where the ETEC C implementation differs functionally from other C implementations. As mentioned, ETEC adheres to the C99 and TR18037 specifications and it is believed to be compliant in these cases. While for some of these issues ETEC will issue errors or warnings when detected, in other cases ETEC cannot tell that there is a potential problem. We believe this is a comprehensive list, but let us know if you are aware of any other cases.

Int-Fract Multiplication

It is typical in automotive crank functions to use fixed-point _Fract type variables to help calculate crank tooth acceptance windows and detect missing teeth. Unsigned _Fract variables represent values from 0 to 1 (actually [0, 1)). The typical calculation in legacy source code might look something like:

```
unsigned int24 Tooth_Period, Half_Window_Width;
unsigned fract24 Windowing_Ratio_Normal;
/* misc. code ... */
Half_Window_Width = Tooth_Period * Windowing_Ratio_Normal;
```

The problem is that according to the TR18037 specification, when an int and a _Fract type are multiplied the result is supposed to be the *fractional* portion of the result, not the integer portion. The code above assumes that the multiplication result is the integer portion. ETEC complies with the TR18037 specification and so if ToothPeriod were 1000 and Window_Ratio_Normal the _Fract value representing 0.5, then ETEC would correctly get 0 as a result (the fractional portion of the result of 500). In order to get the expected result the code should be modified to use the appropriate fixed-point library call, in this case:

```
Half_Window_Width = muliur(Tooth_Period, Windowing_Ratio_Normal);
```

This would yield the value of 500 in Half_Window_Width given the example numbers above. ETEC detects an implicit conversion of a _Fract to an int type, above an int-fract multiplication and issues an error. If the user really does want the code get the fractional result and convert it to an int they must use an explicit type cast. For the list of all supported fixed-point library calls the reader should check the etpu_lib.h header file.

Signed Issues

There are several areas where operations on signed integer variables can get different results between ASH WARE and other tools. One case is with regards to bit-fields. ETEC treats “int” bit-fields as “signed int” bit-fields. Thus, unless it is desired to store signed values in bit-fields,

it is recommended that they be declared “unsigned int” (or unsigned short, unsigned char). Otherwise unexpected results could occur and the generated code is much less optimal as it has to potentially perform sign extension.

```
BitfieldStruct.FourBits = 0xf; // int FourBits is 4 bits wide
int x = BitFieldStruct.FourBits; // x = 0xffffff
```

Another area of problems with signed integers is division and modulus operators. The underlying hardware unit performs unsigned division/modulus only. Other tools may perform unsigned division even if the operands are signed. For example, it would yield a result of 0 for 1000 / -2 since -2 is 0xfffffe in 2’s complement. ETEC will correctly yield -500 for this expression. However, this comes at a significant hit in code performance. Therefore unless signed division or modulus is truly needed, it is highly recommended unsigned types be used, or at least a type cast done to unsigned in order to get tighter code.

The last place to be careful is with regards to sign extension. When mixing operands of different sizes and some of the operands are signed ETEC may perform sign-extension in places where other tools do not. Below is one example case.

```
unsigned int24 z, x = 0x800;
int8 y = 0x80;
z = x | y; // z gets 0xffff80 !
```

To correct the problem y should either be declared as unsigned int8, or should be converted to unsigned int8 before the bitwise-OR. In general with the eTPU, it is best to declare integer variables as unsigned integers unless they truly need to hold signed values.

Bitfields & _Bool Types

With ETEC, bitfields can only be defined with base types of char (int8), short (int16) or int (int24), signed or unsigned. Other tools allow bitfields in int32 types, however, unless the user is careful to make sure a bitfield does not cross the bit 23:24 boundary, incorrect code can result. So if existing code contains a construct like:

```
struct BF {
   unsigned int32 a : 24;
   unsigned int32 b : 8;
};
```

The source should be changed to:

```
struct BF {
   unsigned int8  b : 8;
   unsigned int24 a : 24;
};
```

The ETEC compiler defaults to allocating only a single bit for channel frame _Bool variables. These _Bool variables get packed into 8-bit units. Note that with ETEC global _Bool variables consume an entire 8-bit unit and are located in the LSB, in order to ensure proper link behavior. The ISO/ANSI standard essentially requires _Bool type variables to each consume 8-bits. In

order to get compliant behavior in ETEC the "–ansi" option can be specified, but this is not recommended for typical eTPU code as it also results in much lower performance.

Enum Type Size

eTPU tools do not constrain enum types to one size, but rather sizes the type based upon the range of enumeration literals of the type. ETEC also does this and fits enums in 1 byte if possible, otherwise 3 bytes. Since it cannot be guaranteed that the algorithm to determine the type size is the same in all compilers, there is some chance that the generated code may contain a size mismatch (between compilations from different tools). This does not affect the eTPU code itself, but does potentially have implications for host-side code and eTPU Simulator script files. Users should verify that enum variables are the size they expect when converting code to use ETEC, and if there is a difference the appropriate changes must be made.

32-bit Data

The eTPU ALU and MDU natively support 24-bit arithmetic, not 32-bit. The architecture does allow 32-bit data to be loaded and stored from memory, and this is all that ETEC supports at this time. If ETEC detects any other operation types on 32-bit data it will issue a compilation error. Other compilers may allow 32-bit data to be used in some places that ETEC won't, without compilation error; for some of these cases the generated code would fail if any bits in the upper 8 are non-zero. Note: with the release of 2.10, ETEC now supports typecasts to 32-bit types.

Pointer Arithmetic

It appears that eTPU toolchains may not always generate compliant code for pointer arithmetic. When a pointer to int24 is incremented by one the underlying value should increment by 4 since that is the stride size for the int24 type. In some cases this works (examples below from Freescale QOM code):

```
Current_Match_Ptr++ ; /*increment pointer*/
```

The code above correctly increments the address in Current_Match_Ptr by 4 bytes. However, elsewhere in the same code:

```
Current_Match_Ptr = Table_Start_Ptr + 4; /*increment pointer*/
```

This generates code wherein the address stored in Current_Match_Ptr is 4 bytes greater than Table_Start_Ptr. Compliant code would add 16 bytes (ETEC). Care must taken in places where non-compliant pointer source code has been written previously.

Named Register Variables

In some cases variables are declared with a specific register type using the legacy "register_<name>" syntax. "register_<name>" is supported via typedefs, but care must be taken declaring such variables, especially when using general purpose registers or the load/store registers. With ETEC, when such a variable is declared the compiler is essentially locked out from using the register in its code generation; this can result in compilation failures if a needed register is not available. An exception is the p register; a named register variable in p acts only as an alias and does not lock out the compiler from using p.

Note that named register variables are treated like they are static even when they have function scope – that is, their state is not saved when a function is called. On the other hand, this does mean that tricky code that passes data to functions via named registers can be made to work (but is generally not desirable).

Volatile

The ETEC C Compiler is highly optimizing, and one of the things that it will optimize is redundant loads & stores to memory. Thus the memory allocated to global and channel frame variables may not be read from / written to each time such a variable is referenced. If this is an issue for a particular variable, it should be declared `volatile`. However, the volatile keyword should only be used where it is strictly needed as it hinders tight code generation.

Labels

Labels have function scope; that is they cannot be referenced outside the function they are defined in. In some tools it appears that labels have global scope and code has been written that references labels in other functions. Such code will not compile under ETEC. Also, per the C specification a label must be followed by a statement, even if it is a null statement (";").

Entry Table Conditions

The ETEC compiler supports "lsr" in the entry condition expression, but not "link". "link" is a write-only register for generating links, whereas "lsr" represents the link service request flag, which is what function entry is actually based upon. Any references to "link" in the entry conditions must be changed to "lsr" in order to compile under ETEC.

Intrinsic Functions

ETEC supports a set of intrinsic functions that gives users access to the specialized eTPU ALU/MDU hardware at the C code level. The full set of intrinsics is described in the reference manual, and in the eTpu_Lib.h header file. Below is an example of some code from the Freescale stepper motor function that becomes far more optimal through the use of an intrinsic:

```
      /* Rotate the pin sequence */
      if ((flags & SM_DIRECTION) == SM_DIRECTION_RIGHT)
      {
#ifdef __ETEC__
        // ETEC code
        pin_sequence = __rotate_right_1(pin_sequence);
#else
        // original code
        /* rotate right by 1 bit */
        pin_sequence >>= 1;
        if (CC.C == 1)
        {
          pin_sequence += 0x800000;
        }
#endif
```


Inline Assembly

ETEC supports inline assembly as of release 1.20A. Details of porting inline assembly are provided in a separate inline assembly porting guide; please reference that for details. However, there may be many cases where the inline assembly is no longer necessary due to ETEC's rich set of intrinsics, better coherency support, and reliable code generation. One example can be found in Freescale's Quadrature Decoder function that is part of sets 3 and 4. Rather than use inline assembly to perform an absolute value, this can be handled easily with an ETEC intrinsic:

```
#ifdef __ETEC__
        // ASH WARE ETEC code
        if(__abs(pc)>=pc_max)
#else
        // original code
        /* if(abs(pc)>=pc_max) */
        register_p  p;
        register_ac a;
        p = pc;
        #asm( alu a = abs(p). )
        if(a>=pc_max)      // if(abs(pc)>=pc_max)
#endif
```

In other cases, inline assembly was used to work-around compilation problems. One such example is in the Freescale stepper motor function. The code snippet below shows the working ETEC C code that replaces the inline assembly work-around put in place originally.

```
#ifdef __ETEC__
    // in order to get integral portion of int-fract multiply, must use one
    // of the muli library functions...
    tmp = muli24ur16(start_period, (unsigned fract16) (accel_tbl[accel_tbl_index-2]));
#else
    /*tmp = start_period * (unsigned fract16)(accel_tbl[accel_tbl_index-2]);*/
    tmp = accel_tbl_index-2;
    tmp = accel_tbl[tmp];
    #asm( alu a = d;  ram p <- start_period. )
    #asm( mdu p fmultu a (16). )
    #asm( SM_L: if mbsy == 1 then goto SM_L, flush. )
    #asm( alu d = mach. )
/* Comments to ETPUC:
   - this expresion is not compiled correctly:
     1. 16-bit unsigned fractional multiplication (instruction fmultu(16))
        is not supported.
     2. overcasting does not works properly. */
#endif
```

There are times when inline assembly is necessary and appropriate. The reference manual and the inline assembly porting guide have more details on the ETEC inline assembly support.

ETEC Currently Unsupported Constructs

This section is a placeholder for unsupported language constructs that are typically seen in eTPU applications. As of release 1.20A there are no major unsupported constructs. There are some minor items; comprehensive lists can be found on the ASH WARE website in the ETEC release notes area.

Simulator Scripts and Host Code

By default, ETEC automatically outputs all data necessary for the host interface into the _defines.h, _struct.h, _idata.[ch], and _scm.[ch] data files. The automatic generation that ETEC can do is output in a well-defined manner, and the macro names generated are well-defined (see the reference manual for exact details)… but the result is that they will almost always be different than what was previously output using other techniques. Thus modifications are required if using this option; the section "Conversion to the Auto-Defines File" describes this process. Note: generally previously used host interface techniques will work as-is.

Another significant difference is that by default ETEC uses a stack-based approach to handling local variable overflow and function calls. Why does ETEC use a stack? The main reason is that that existing eTPU code that has local variable overflow or uses function calls potentially faces a major bug when used on a dual eTPU (e.g. MPC5554). If the same code is run on both eTPUs, and that code has function calls or local variable overflow, the global "scratchpad" storage for these variables / calls can conflict and nasty data corruption problems can ensue. The second reason is that the stack-based approach can actually be more efficient with regards to total shared data memory (SDM) usage. Note that eTPU code that does not have overflow local variables, or does not make function calls (or at least function calls that do not require the stack), does not actually use the stack and can potentially not allocate any memory for it. Finally, the stack-based solution is a more compliant and robust solution.

That being said, ETEC does offer a "scratchpad" based programming model wherein local variables and function call state that overflow registers are placed into a "scratchpad" memory. In code where there is significant local variable overflow or function call state, there are cases where using this model can result in more efficient code, although as mentioned above, care must be taken, and the end result can be increased memory usage. The scratchpad can either be placed in the global memory (-globalScratchpad compiler option), or engine-relative memory (-engineScratchpad compiler option; only available on the eTPU2). The latter option does prevent dual-engine corruption problems, but results in even more Shared Data Memory (SDM) usage.

Conversion to the Auto-Defines File

As mentioned, although ETEC *automatically* generates virtually all the information needed for the host to interface with the eTPU, it will very likely not generate the very same macro names currently specified within a user's #pragma writes. Also, there is one main thing that does not end up in the ETEC "defines" file – eTPU-side macros that are in turn exported by #pragma writes into new macros. The best approach on these is to break them out into their own header file that can be included in both the eTPU code and the host-side code. So for the Freescale UART function, this header might be called "etpu_uart_common.h" or similar, and it would contain

```
/* Host Service Request Definitions */
#define FS_ETPU_UART_TX_INIT 4
#define FS_ETPU_UART_RX_INIT 7

/* Function Mode Bit Definitions - polarity options */
```

```
#define FS_ETPU_UART_NO_PARITY 0
#define FS_ETPU_UART_EVEN_PARITY 0  + 2
#define FS_ETPU_UART_ODD_PARITY 1  + 2
```

Everything else that is needed is auto-generated in the UART case. One way to go is to create a conversion header file to convert the ETEC output macros to the macro names currently in use. The benefit of this is existing host code and simulator scripts require far fewer changes. For UART, such a file (etec_to_etpuc_uart_conv.h) might look like:

```
#ifndef __ETEC_TO_ETPUC_UART_CONV_H
#define __ETEC_TO_ETPUC_UART_CONV_H

#include "etpuc_set1_defines.h"

/* Function Configuration Information */
#define FS_ETPU_UART_FUNCTION_NUMBER            _FUNCTION_NUM_UART_
#define FS_ETPU_UART_TABLE_SELECT               _ENTRY_TABLE_TYPE_UART_
#define FS_ETPU_UART_NUM_PARMS                  _FRAME_SIZE_UART_

/* Parameter Definitions */
#define FS_ETPU_UART_MATCH_RATE_OFFSET          _CPBA24_UART_FS_ETPU_UART_MATCH_RATE_
#define FS_ETPU_UART_TX_RX_DATA_OFFSET          _CPBA24_UART_FS_ETPU_UART_TX_RX_DATA_
#define FS_ETPU_UART_BITS_PER_DATA_WORD_OFFSET  _CPBA8_UART_FS_ETPU_UART_BITS_PER_DATA_WORD_
#define FS_ETPU_UART_ACTUAL_BIT_COUNT_OFFSET    _CPBA8_UART_FS_ETPU_UART_ACTUAL_BIT_COUNT_
#define FS_ETPU_UART_SHIFT_REG_OFFSET           _CPBA24_UART_FS_ETPU_UART_SHIFT_REG_
#define FS_ETPU_UART_PARITY_TEMP_OFFSET         _CPBA8_UART_FS_ETPU_UART_PARITY_TEMP_
#define FS_ETPU_UART_RX_ERROR_OFFSET            _CPBA8_UART_FS_ETPU_UART_RX_ERROR_

#endif // __ETEC_TO_ETPUC_UART_CONV_H
```

On the right are the macros created by ETEC, which are set in the included defines file. This file (etec_to_etpuc_uart_conv.h) can then be included in host code or simulator scripts to translate ETEC output to the old naming convention. See the reference manual for details on all the items output into the ETEC defines file, and the algorithms used to generate the names.

As a side note, many simulator script files have been seen that have commands like:

```
write_chan_entry_condition( PWM0, 1);       // Set entry table to alternate.
write_chan_data32 (PWM0,  0, 1000000);      // Period
```

Needless to say this is an “accident waiting to happen”. If the entry table type were to change, or the Period channel frame variable offset to change, the above would no longer work (same applies to host code). The auto-generated macros should always be used:

```
write_chan_entry_condition( PWM0, _ENTRY_TABLE_TYPE_PWM_);
write_chan_data24 (PWM0, _CPBA24_PWM_Period_, 1000000);
```

More recent versions of the compiler support exporting use macros directly into the auto-defines file. The following lines in eTPU source:

```
#define TEST_INIT_HSR 7
#define TEST_STR "xyz"
#pragma export_autodef_macro "ETPU_TEST_INIT_HSR", TEST_INIT_HSR
#pragma export_autodef_macro "TEST_STR", TEST_STR
```

Results in the following in the auto-defines file:

```
// exported autodef macros from user "#pragma export_autodef_macro" commands
#define ETPU_TEST_INIT_HSR  7
#define TEST_STR  "xyz"
```


As always, see the compiler reference manual for more details.

Stack Initialization

If utilizing the default stack programming model, each eTPU requires its own stack, so a dual eTPU requires 2 unique stacks to be allocated. On a dual eTPU the shared data memory (SDM) layout may end up looking something like

global variables start at address 0

engine 0 stack base

engine 1 stack base

engine 0, channel 0 channel variables
engine 0, channel 1 channel variables
engine 0, channel 5 channel variables
….
engine 1, channel 0 channel variables
engine 1, channel 3 channel variables
….
engine 1, channel 27 channel variables

Unfortunately, this also means that the stack base cannot be stored as a global because code may run on both eTPUs simultaneously. Instead, each eTPU function that needs to use the stack gets an additional __STACKBASE channel frame variable allocated. The stack can be sized using the ETEC defines file _STACK_SIZE_ macro (NOTE: if _STACK_SIZE_ is 0, the code uses no stack whatsoever and no stack initialization is required). This will be the statically computed maximum stack usage value for the compiled code, although at the moment it is a hardcoded value. Stack initialization can be done as follows in a simulator script; a similar approach could be used within host code.

```
#define ETPU_A_STACK_BASE _GLOBAL_DATA_SIZE_
#define ETPU_B_STACK_BASE ETPU_A_STACK_BASE + _STACK_SIZE_
#define CHANNEL_FRAME_BASE_ADDR (ETPU_B_STACK_BASE + _STACK_SIZE_ + 7) & ~7
// only initialize the stack base for each channel IF the channel's
// function requires the stack (detected by whether a STACKBASE
// macro exists for the function)
#ifdef _CPBA24_TOOTHGEN___STACKBASE_
write_chan_data24 (TOOTHGEN_CRANK_CHAN, _CPBA24_TOOTHGEN___STACKBASE_,
ETPU_A_STACK_BASE);
#endif
#ifdef _CPBA24_Crank___STACKBASE_
write_chan_data24 (CRANK_CHAN, _CPBA24_Crank___STACKBASE_, ETPU_A_STACK_BASE);
#endif
#ifdef _CPBA24_Cam___STACKBASE_
write_chan_data24 (CAM_CHAN, _CPBA24_Cam___STACKBASE_, ETPU_A_STACK_BASE);
```

```
#endif
```

Note that the next generation eTPU (eTPU2) has an engine-relative addressing feature. This may allow the stack base settings to be moved from channel frames to engine-relative space, potentially reducing SDM usage and simplifying the initialization process. Or it may allow other compilation options. TBD. Note that existing stack mechanism generates working code on all eTPU parts.

Data Initialization

Global and channel frame (and if used, engine-relative on the eTPU2) data initialization can be accomplished using the auto-generated <>_idata.[ch] files. It is fairly unusual in eTPU code to have static initialized channel frame data, but it is supported with ETEC. The <>_idata.c file contains initialized arrays that host code can link with and use with memcpy() or similar to perform the initialization. Alternatively the <>_idata.h file can be use with its macro-based approach; the <>_idata.h file must be used with simulator scripts. An example of a simulator script using an idata.h file is shown below; first a segment of an idata.h file is displayed, followed by the script file use of it.

```
#ifndef __GLOBAL_MEM_INIT32
#define __GLOBAL_MEM_INIT32( addr , val )
#endif

// macro name ( address_or_offset , data_value )
__GLOBAL_MEM_INIT32( 0x0000 , 0x00000777 )
__GLOBAL_MEM_INIT32( 0x0004 , 0x55ffffff )
__GLOBAL_MEM_INIT32( 0x0008 , 0x00000000 )
__GLOBAL_MEM_INIT32( 0x000c , 0x00000000 )
// Test Channel Frame Initialization Data Macros

#ifndef __Test_CHAN_FRAME_INIT32
#define __Test_CHAN_FRAME_INIT32( addr , val )
#endif

// macro name ( address_or_offset , data_value )
__Test_CHAN_FRAME_INIT32( 0x0000 , 0x80000000 )
__Test_CHAN_FRAME_INIT32( 0x0004 , 0x0d000000 )
__Test_CHAN_FRAME_INIT32( 0x0008 , 0x4d000000 )
__Test_CHAN_FRAME_INIT32( 0x000c , 0xff000000 )
```

The simulator script uses the above as follows

```
// load the initialized data
#undef __GLOBAL_MEM_INIT32
#define __GLOBAL_MEM_INIT32(address, value) *((ETPU_DATA_SPACE U32
*) address) = value;
#include "DeclTest_B_idata.h"
#undef __GLOBAL_MEM_INIT32
```



```
#define TEST_CFBASE_0 0x100
#define TEST_CFBASE_1 0x200
#undef __Test_CHAN_FRAME_INIT32
#define __Test_CHAN_FRAME_INIT32(offset, value) *((ETPU_DATA_SPACE
U32 *) TEST_CFBASE_0+offset) = value;
#include "DeclTest_B_idata.h"
#undef __Test_CHAN_FRAME_INIT32
#undef __Test_CHAN_FRAME_INIT32
#define __Test_CHAN_FRAME_INIT32(offset, value) *((ETPU_DATA_SPACE
U32 *) TEST_CFBASE_1+offset) = value;
#include "DeclTest_B_idata.h"
#undef __Test_CHAN_FRAME_INIT32
```

to initialize global data and to initialize two channels of function “Test”.

Note that for both initialized data, and for shared code memory initialization discussed in the next section, the default is to output the data in 32-bit chunks. The “-data8” option can be specified to the linker in order to generate 8-bit initialization both the *_idata.[ch] and *_scm.[ch].

Shared Code Memory Initialization

ETEC generates a <>_scm.[ch] files for inclusion in host builds. The *_scm.c file contains an actual array definition, while the *_scm.h file contains an array data presenting the generated eTPU code image. The *_scm.h file can be included in an array declaration of the user’s choosing if the default one in the *_scm.c file is not usable.

```
// SCM - Static Code Memory, Memory which the eTPU executes (eTPU + 0x10000)
unsigned int _SCM_code_mem_array[] = {
#include "etpuc_set1_scm.h"
};
```

Contents of the associated *_scm.h file:

```
/*0x000*/ 0x42BC42BC, 0x42BC42BC, 0x42BC42BC, 0x42BC42BC,
/*0x010*/ 0x42BC4200, 0x42BB42BB, 0x42BA42BA, 0x42BA42BA,
/*0x020*/ 0x42BC42BC, 0x42BC42BC, 0x42BC42BC, 0x42BC42BC,
/*0x030*/ 0x42BB42BB, 0x42BB42BB, 0x42BA42BA, 0x42BB42BB,
/*0x040*/ 0x02BF02BF, 0x02BF02BF, 0x02BF02BF, 0x02BF02BF,
...
```

Auto-struct Host Interface File

For additional host interface support, ETEC also generates a <>_struct.h file that contains struct declarations that can be used by host code to overlay eTPU data memory for simplified access to shared host/eTPU data. C struct declarations are auto-generated for global data, engine data (if needed, eTPU2 only), and all channel frames.