Freescale Semiconductor
Application Note

AN2848
Rev. 0, 09/2004

Programming the eTPU

by: Mike Pauwels
TECD Systems Engineering

This is one part of a series of application notes intended to help the microcontroller systems engineer to design and implement code for the Enhanced Time Processor Unit (eTPU). The notes are intended to suggest design strategies, map techniques for common classes of problems, and steer the user away from difficult operations and algorithms. This first note provides an overview of the programming project, with suggestions on how to make partitioning and top level design decisions and avoid common pitfalls and problems.

Table of Contents

RESOURCES

Websites

www.ashware.com

www.bytecraft.com

www.eTPU.com

Training

Programming the eTPU

Presented by Ash Ware and Freescale. See www.ashware.com.

Publications

eTPU Reference Manual

freescale™ semiconductor

Byte Craft eTPU C Compiler User's Manual

1 Overview

The eTPU is an autonomous slave processor offered on various families of Freescale microcontrollers. It is an enhanced version of the TPU, which has enjoyed one of the longest successes of any microcontroller peripheral. Despite its popularity in specific markets, widespread adoption of the TPU was hindered by a lack of high-level language support and limited availability of development tools.

The eTPU was designed from the start to be supported by a high-level language compiler, and as such has become accessible to and adopted by a very wide range of customers even before the silicon had been qualified. Many of these customers had previous TPU experience, and approaching the eTPU with an understanding of the limitations of this type of device, found that they were pleasantly surprised at the increased capabilities of the eTPU. However, the great expansion of resources inspired some of these users to add features until they finally overwhelmed the capabilities of the device. Other users, never having worked within the tight constraints of the TPU, discovered the limitations of the device trying to implement overly ambitious designs approaches, and were obliged to scale back or even restart their plans. This application note will offer some guidelines to help users streamline their design process without running headlong into resource limits.

2 Architecture

The eTPU is a slave co-processor tightly coupled to up to 32 I/O channels, each associated with an input and an output signal; see Figure 1. The eTPU processing engine executes code from a code memory, using parameters in a data store which is simultaneously accessible to the host.

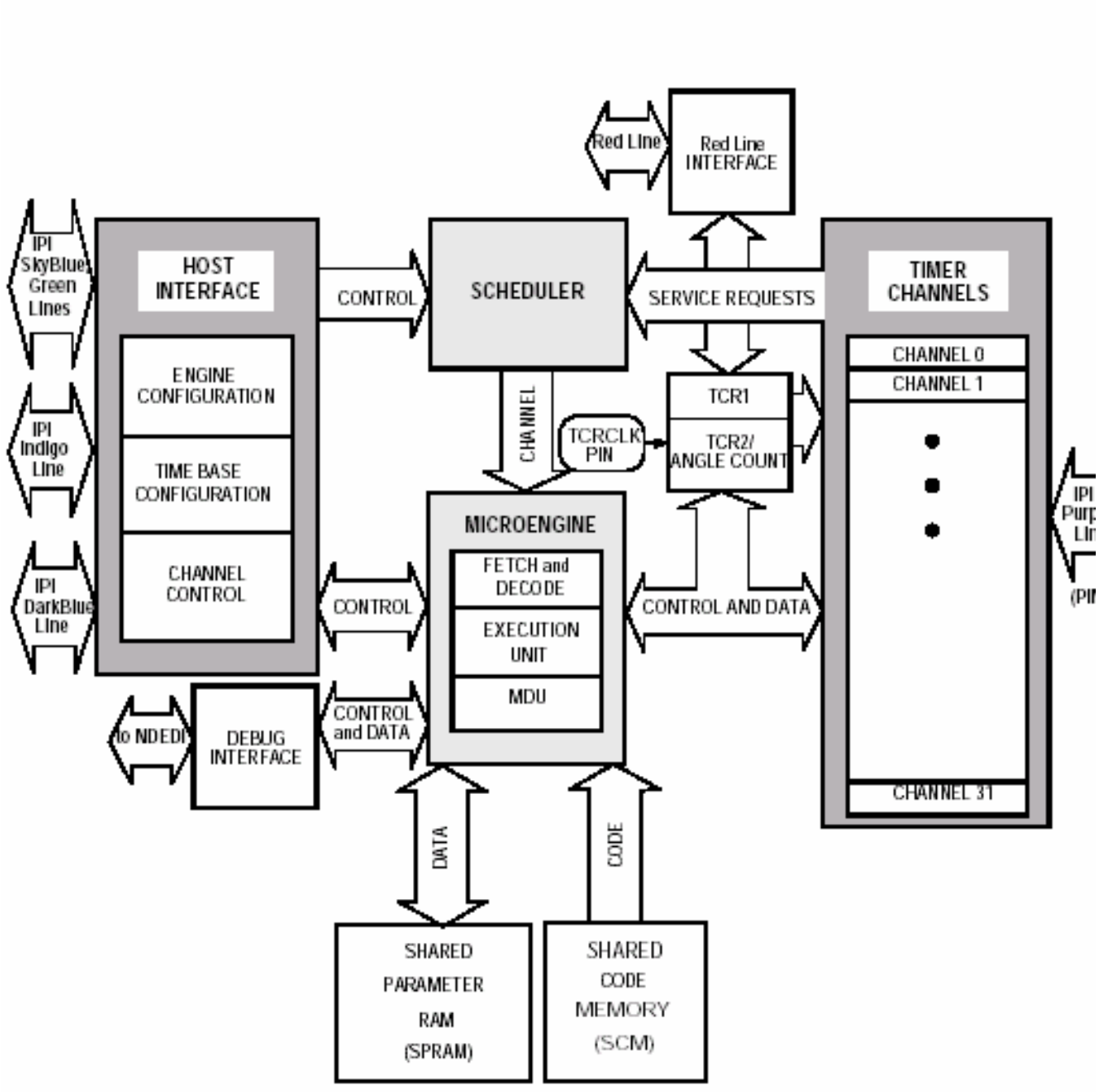

Figure 1. eTPU Block Diagram

The input/output channels of the eTPU each have a pair of Match and Capture units interfaced to one of two timer/counter (TCR) registers. Logic in the channel enables the hardware to detect or drive pin transitions with a high degree of timing precision. Details of the channel architecture is a topic for a subsequent note, but a block diagram is given in Figure 2.

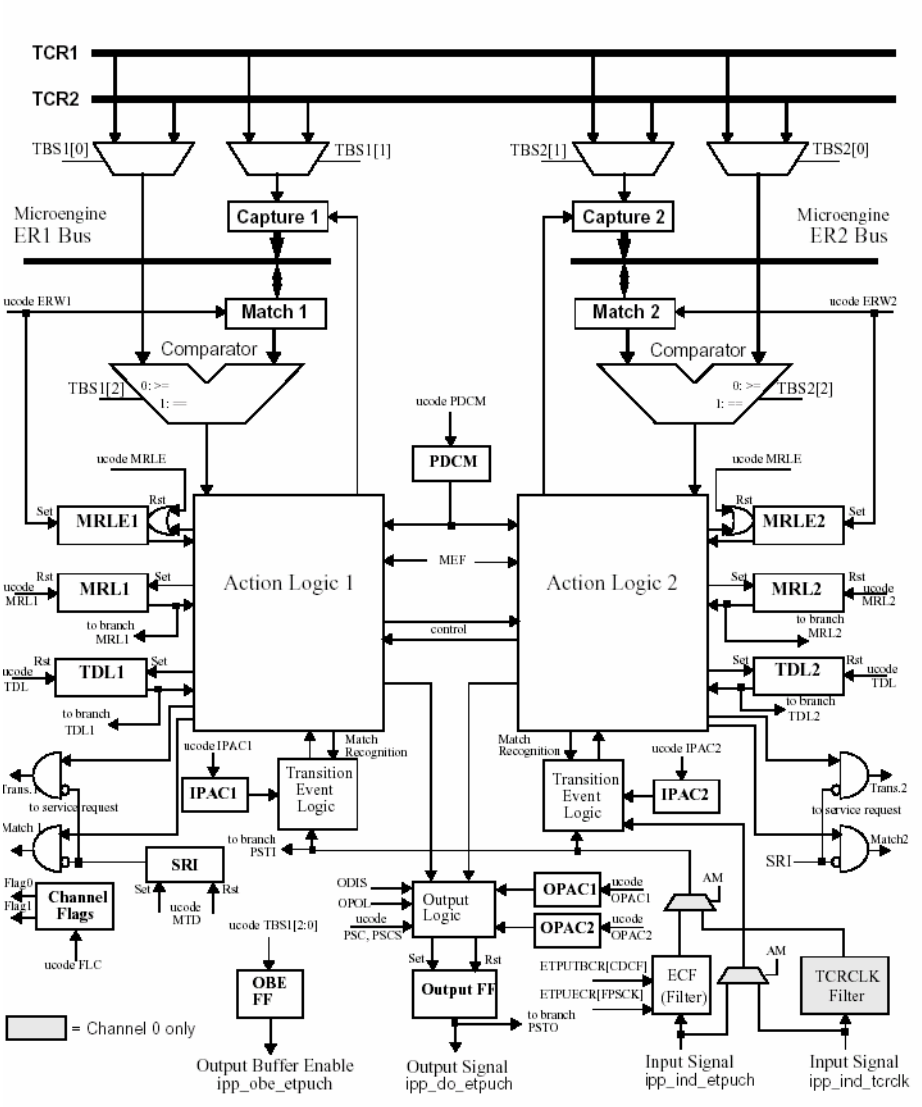

Figure 2. Channel Block Diagram

The hardware associated with each channel is controlled by a number of registers, which determine actions associated with the input and output pins and eTPU service requests. These registers are accessed by a special purpose eTPU engine that can be programmed in C. The engine has a program store for the engine software and a data store for function parameters. The memory blocks, whose size varies with the MCU, are designed to be shared by two eTPUs, as well as being accessible to the host. Software threads in the engine can be started by channel actions, and very complex control systems can be implemented that, once started, can operate independently of the host.

Host action is required to setup the functions in the eTPU engine and channels, and the host may be incorporated to a greater or lesser extent in the closing of an eTPU control loop. The host controls the eTPU by writing the function code into the control store, writing the operating parameters into data store, and configuring the basic functionality of the individual channels.

Let's look at how this architecture works by considering a classic example; see Figure 3. If the user wanted a Pulse Width Modulated (PWM) output signal on one of the channel pins, he would need a TCR and comparator to cause the output to transition high at some point in time. When that transition occurs, he would want another transition to be scheduled at a time equal to the time of the first transition, plus a high time determined by his software.

The first transition will cause a request for the engine to service the channel. The engine executes an instruction thread, which has been loaded by the host into the program memory, to calculate the high time for the pulse. The parameters for this calculation would also have been provided by the host in the data store. The engine would set up channel hardware to make a low transition when the high time expires, at which time another service request can be issued to the engine to service the low time. If the low transition executes a similar instruction thread, the process can continue forever without real time intervention by the host. If the host changes the operating parameters in the data store, the PWM can be modulated.

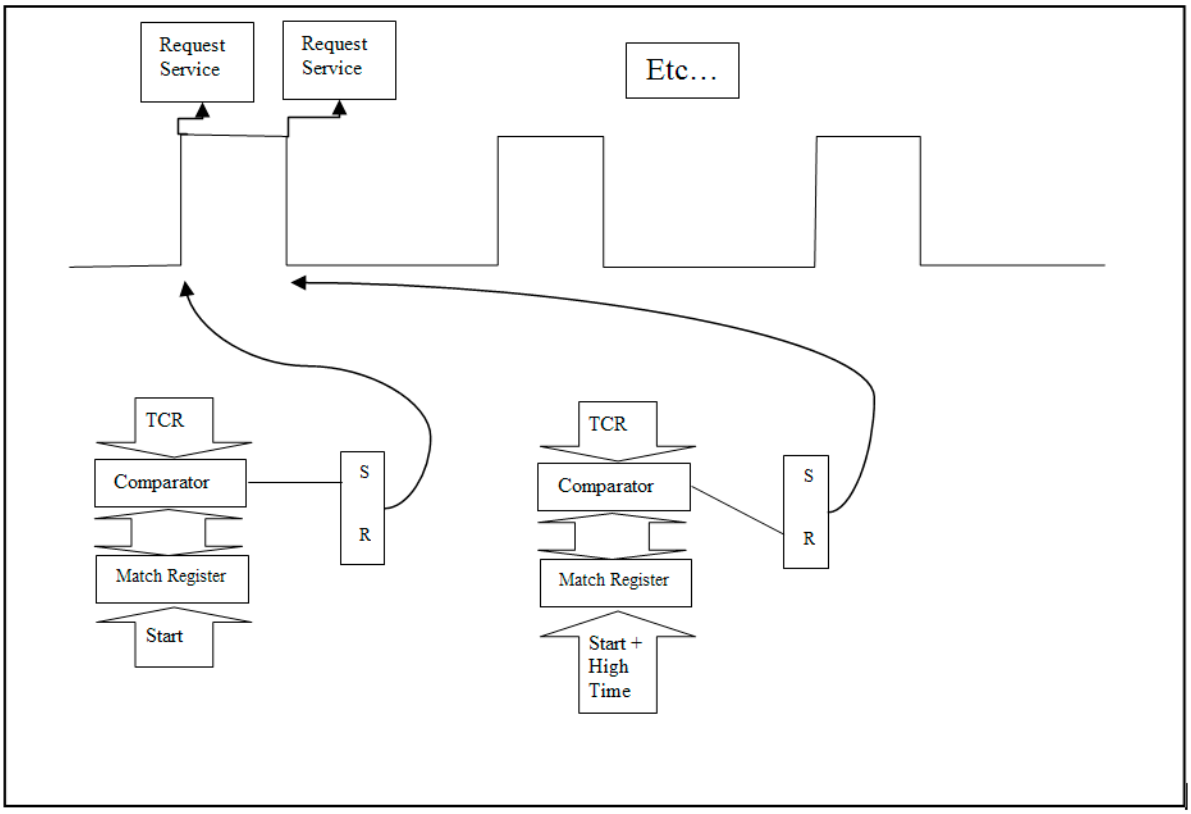

Figure 3. Controlling a PWM

If PWM were the most complex function required of the eTPU, this note would not be needed. In fact, systems have been implemented in automotive engine control where the eTPU detects the engine position and speed, controls fuel, spark, transmission shifting, and emission control valves, and drives several additional sensors and actuators. Fuel pulses have been designed to modulate the distribution of fuel in the cylinder, and multiple spark pulses are generated to ensure complete, clean burn of the fuel. In other eTPU applications, communication pulses have been produced, and complete closed loop motor control

systems implemented. Along the way, a number of difficult problems posed by the unique nature of the eTPU have been discovered and solved. This application note contains some of the wisdom of prior experience for the benefit of new engineers.

3 Function Design – Hardware

The first step in applying the eTPU to a timing and control problem is to determine if the application fits within the capabilities of the eTPU. To do this requires a working understanding of the architecture of the timing channels. A detailed treatment of this will be covered in another application note, but here is a summary of the general capabilities and limitations of the channel hardware:

- RESOLUTION – The channel hardware can set a pin high or low, or toggle it immediately or at some time in the future. This occurs when a free running counter/timer matches a register value. The size of the register is 24 bits, and the counter can be incremented as fast as ½ the frequency of MCU system clock. This is the finest resolution available for timing.
- MAXIMUM TIME – The timer/counter can be slowed by prescaling, and the TCR range is 24 bits. If the TCR resolution is 100 nanoseconds, then the counter range is about 1.67 seconds. More significantly, signal timing can incorporate software counters, enabling virtually any maximum time length.
- MINIMUM TIME – The channel has two match comparators, and a pulse can be set to start at one time and end at another after only a single tick of the selected counter/timer. Thus, the minimum time pulse that can be produced is 2 times the system clock.
- INPUT LIMITS – The above limits are virtually the same for input transition measurements.
- TIME BASES – While there are two time bases provided, the range of one counter is sufficient to enable virtually any combination of pulse times. The second one can be driven by an asynchronous external source, or be controlled by special Angle Clock circuitry provided in the eTPU. The Angle Clock is a subject of a future note, but in general it is a system where the angle of a spinning shaft can be tracked and the extrapolated angle can be used to time input or output events.
- INTERACTION OF EVENTS – The channel hardware can be configured into a number of operating modes, where, for example, output pulses can be timed by two different time bases, input pulses windowed by timers, or outputs can be used to enable timers all without direct software intervention.
- SYNCHRONIZATION – Since all channels operate from the same timer/counters, software can be used to synchronize inputs and outputs with quite complicated algorithms. For example, a spark pulse can be made to start at a projected time before the firing angle, and fire exactly on the angle, tracking as closely as possible the variation in speed of an engine.
- LATENCY – Since the channels have only two action units for each pin, the eTPU requires software intervention before a third transition can be acted upon. This means that while it is possible to accurately produce or measure a narrow single pulse, a third transition can only be produced or detected after the eTPU engine has begun to service the channel. The minimum time

required by the eTPU engine to service one of the edges and reset the channel for the third edge is highly dependent on the configuration and activity in the rest of the eTPU system. However, even with no other eTPU activity, this time cannot be reduced to less than 10 CPU clock cycles.

The second step is to select a channel mode appropriate for the task. The eTPU provides 13 preprogrammed modes for channel operation. Detailed instructions on selection and using these modes are the subject for a future application note, but the following summary may help in system design:

- When it is necessary to port a function from the TPU, the *sm_st* (SingleMatchSingleTransition) mode is the TPU compatible mode.
- If multiple matches are desired, the engineer must decide how the matches are to be related. The *em_* (EitherMatch) modes allow either match to occur and both matches can request service.
- The *em_* modes can be either *b_* (Blocking) or *nb_* (NonBlocking), which determines whether or not the first match will block the second.
- The *bm_* (BothMatches) modes require both matches to occur before the channel requests service. The both match modes may be further designated *o_* (Ordered), which requires that MatchA occurs before MatchB will be recognized.
- In m2_ (Match2) modes, MatchA does not request service but enables MatchB, after which MatchB e and will block a subsequent MatchA
- All of these modes can be further designated *_st* (SingleTransition) or *_dt* (DoubleTransition). The only difference between these is that single transition will request service after the first output transition and double transition will request service after the second transition. Note that although all modes are either *_st* or *_dt*, the user has the option of disabling the transition on the match for an input-only function.
- There is also an enhanced mode, *sm_st_e*, where a transition can be interlocked by a match.

Details of these modes can be found in the *eTPU Reference Manual*.

4 Function Development – Tools

A C compiler for the eTPU is available from Byte Craft, Ltd. of Waterloo, Ontario. Byte Craft has posted and maintains a web page with frequently asked questions (FAQ) that can be a valuable resource for eTPU users. The URL for the FAQ is:

http://www.bytecraft.com/public/etpuc/downloads/etpuc_faq.chm

The C compiler is designed to be compliant with a proposed ISO Standard for C for Embedded Systems. This standard allows significant new features in the C language which have been found critically important in dealing with the eTPU. The new language and the new eTPU have inspired a number of significant compiler advances found in the Byte Craft product.

- The compiler passes information to the host processor allowing a one-step make process, effectively linking the eTPU functions to the host code. The information passing is enabled through a number of post-processing macros available in the Byte Craft compiler. Details of these macros will be described in a future application note.

- The host compiler and linker do not require special features to use the eTPU information. Information is provided in C compatible files and the subsequent compilation of the host project can provide all the information for code passing and reference resolution.
- The compiler compiles directly to eTPU microcode, often producing one microcode instruction for multiple source instructions.
- The mapping of entry addresses for service requests from the host or the channel hardware is compiled from C statements in the eTPU source.
- The new coding standard allows direct access to registers and other resources in the eTPU engine, enabling the user to write “Assembly C” which provides the data flow analysis of a compiler in a low level, assembly-like language.

Further information on the eTPU_C compiler can be obtained from Byte Craft.

A Simulator for the eTPU is available from Ash Ware, Inc. of Beaverton, Oregon. Ash Ware has had extensive experience with simulation for the TPU, and cooperated during the development of the silicon by co-validating the simulator. Most experienced users find that the Ash Ware stand-alone simulator is the best way to develop their initial eTPU software. When systems require tight coupling of the eTPU and the CPU, a full system simulator is also available for later stages of the project. For further information, please contact Ash Ware.

5 Function Design – Software

Simple functions like the PWM described above do not require an eTPU. By simply reloading a down counter, the hardware can sustain a PWM without service by a programmed engine. Suppose now that the user wants not just a PWM, but a PWM implemented sine wave modulated by a feedback signal. For example, the channel could produce a 40 kHz PWM waveform which is modulated by a 400 Hz sine wave whose amplitude is driven by a control loop parameter. A low-pass filter and amplifier would be all that was required to reproduce the desired 400 Hz signal. This means that the duty cycle of each period would be determined by a sine value times an externally determined amplitude.

While a special piece of hardware could be designed to implement this function, the eTPU can handle it quite well in software. The complete details of such a system are beyond the scope of this application note, but it is instructive to look at the necessary design strategy. The engine in the eTPU could be given the following equation:

```
HighTime =

(Amp * sin(2*PulseNumber/(PWM_frequency/Modulation_frequency))/MaxAmp
```

This works on a spreadsheet, and given a floating point and trig library, the compiler could produce code for it. However, no sensible designer would write the algorithm in this way. The point is that the correct design approach is to look through the requirement at the CPU executing the code and to design a optimal algorithm within the constraints of the system.

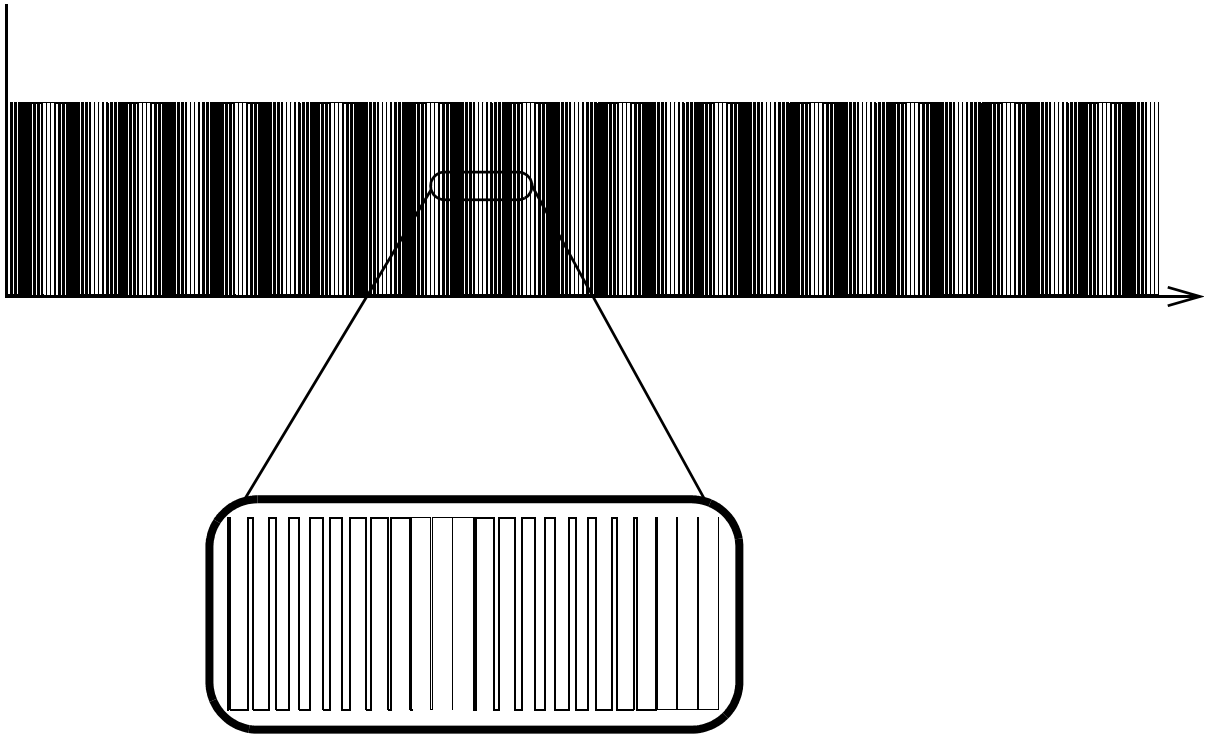

Figure 4. Sine Modulated PWM Waveform

There are three hard limits on the eTPU execution engine: program space, data store, and time. There are methods to trade off between these three, and the strategy must be dictated by the particular system design. However, there is nothing to be gained from unused program or data memory, while every cycle wasted will affect the performance of all of the eTPU functions. Execution time then should be the primary focus for algorithm optimization. If the sine function in the above example could be provided in a lookup table and there is room for the table in the memory, the savings in code space and execution time will be significant.

The strategy used to husband these resources must be dictated by the application, but a few guidelines will help to plan a successful design:

- Make conservative decisions in partitioning the function between the host and the eTPU. If an operation can be placed in either machine, put it in the host. For example, if the requirement is to return the frequency of an input waveform, the eTPU can measure the period directly. The frequency is a scaled reciprocal of the period. When the host requires the frequency, it is trivial to do the math using the more powerful processor. Partitioning the calculation into the eTPU increases the execution time in the eTPU and requires a parameter to store the value.

- The Applications Programmers Interface (API) needs to be designed as part of the eTPU function. An eTPU function cannot be considered complete without the means to initialize and exchange data with the host. Remember that the eTPU is a 24-bit processor while the host and memory system is organized around 32 bits.
- Use 24-bit data types if possible, particularly when indirectly referenced. The eTPU does not have a byte size addressing mode. Significant code is generated by the compiler to dereference byte pointers. An array of (int24) words will take up more data memory, but save significant program space and execution time. Note, however, that the machine can handle arrays of bits quite efficiently.
- The eTPU data memory can be read by the host in two locations. One of these locations returns the 24-bit data automatically sign extended. Accesses by the host to these addresses will not affect the upper 8 bits of the parameter. Using this alternate address space can simplify parameter passing between eTPU and host.
- Reconsider the use of nested subroutines. The eTPU has a single return address register and must store off the value the when a second call is made.
- Use the library functions where possible. They have been designed to balance proper operation with minimum code size. If you find a source code change in a library that reduces the code size, it possibly has an undesirable side effect. Proceed only if you understand the consequences.
- Reconsider parameter-driven channel configurations. The TPU had a convenient instruction: *config := p*. The eTPU does not an equivalent instruction, mainly because the channel configuration options are much more numerous than in the TPU. If you need the option to invert your pulse from high going to low going, implement that option. However to replicate the original *config* subinstruction has been found to be expensive on the eTPU.
- Be careful about auto variables (in the current version of the compiler). These are the non-static temporary variables declared within a function. A problem arises when the function can be instantiated on both eTPU engines. The compiler will allocate an absolute address for these variables, and if one instance of the function is running on each engine simultaneously, there is a possible collision in accessing the variable and consequent corruption of the data. If a function can be instantiated to run on either engine, the only work around at this time is to declare all local variables as *static*. A compiler extension is currently being reviewed to correct this problem.
- Another compiler extension is under construction to modify the scoping of variables for groups of functions. When a user tries to group a number of channels to perform one coordinated function on several pins, for example an H-driver, and then instantiates a number of these groups, the C language does not provide a convenient scoping for the variables local to a group. The Byte Craft eTPU_C compiler will provide an extension to handle this situation. Details of the extension will be provided in a later note.
- Whenever a thread is entered or the CHAN register is written by the eTPU code, the ERTx registers are updated from the channel capture registers. Since these registers are also used to read and write the match registers, the user should take care in all accesses of the channel match and capture registers.

- Optimize your source. The compiler has a sophisticated system of optimizers, but it cannot always rewrite inefficiently written code. We will try to provide specific examples of this in a separate application note.

The design of an eTPU function should follow some general guidelines for best results:

- The compiler recognizes code for the unique architecture of the eTPU when it encounters the following statement:

```
#pragma ETPU_function name standard/alternate @functionnum
```

- The function is divided into threads which are distinguished by the source of the service request that initializes the thread. The thread entries are established by a string of *if…else if…else if……else* statements. The conditions in the if statements must uniquely define one or more entry conditions according to whether the standard or alternate entry table is selected in the *#pragma* statement. All entries in the table must be provided for. All entries not explicitly covered by the if statements will be directed to the trailing *else*.One thread, normally entered by an initialization HSR, sets up the channel for the function. This includes selecting the channel mode, the TCRs to be used, and the initial input and/or output pin actions.
- If the channel is to be used to request service, the thread must enable channel service requests.
- In threads which are entered in response to channel or link service requests, the flag requesting the service should be cleared before exiting the thread. If the flag is not cleared, the channel will immediately be rescheduled for service when the thread ends. Note that exiting a thread (executing an *end*) will automatically clear the HSR bits.
- Use the Simulator for debugging and analyzing your code. In the situation described in the previous bullet, the simulator will provide a warning if a channel constantly requests service due to an un-cleared flag.

6 Host Interface Design

The eTPU is a peripheral device within a microcontroller and is completely controlled by the host CPU. The system engineer needs to provide a number of setup and control functions for eTPU operation. Details of these operations are given in the Reference Manual and in the example functions. The operations include the following steps:

- Write the code image to the eTPU Shared Code Memory.
- Configure the MCU ports as required to enable the selected eTPU pins.
- Set up the global eTPU registers. This includes pin filter control and Timer/ Counter Register prescaler and control. Global interrupts may be enabled if necessary and the MISC can be setup at this time, although it is not necessary to use the MISC during function development.
- Setup the function that is assigned to each channel. This includes writing the compiler generated function number and the initial parameters to the local function frame for the channel. The compiler can provide all addresses automatically.
- Select a high, medium, or low priority for each function. Note that this operation allows a function to be scheduled and execute service threads. The only certain means for the host to disable an eTPU channel is to set the priority to zero.

- Issue an initialization Host Service Request (HSR) to the channel. Note that a service request is necessary for an eTPU function thread to execute. No eTPU operation can occur until at least one HSR is issued. However, depending on the eTPU software, once a channel is started, the eTPU software can schedule new service threads without additional HSRs from the host. Also, it is possible for a software thread on one channel to cause a service request on another channel.
- Interact with the eTPU threads according to the system design. This can involve simple reading or writing parameters, or coordinated interrupt handling and service requests.

7 Simulating the eTPU Function

When a function under development is simulated on the Ash Ware stand-alone simulator, the simulator script files take the place of the host CPU. The simulator is a powerful development tool which is seldom used to its fullest capacity. However, after an hour or two in the training class, most users become sufficiently proficient to analyze and debug common designs. Here are some of the more useful features of the stand-alone simulator:

- Setting up one or more channels using commands similar to your host code. This includes interconnecting channels of the eTPU to test more complex systems.
- Providing external input pin stimuli through a script file. The pin transition can be timed to simulate a number of external inputs, and can be interleaved with “host” operations in the script file.
- Displaying output pin action against time in a logic analyzer window. Hint: if you are having trouble finding your output signal, try checking the “Auto Scroll” button.
- Reading and modifying memory and watching source symbolic values.
- Breakpointing the script or the code of any function.
- Reading or modifying the contents of internal registers.
- Stepping through the code by instruction, source line, or thread.
- Tracing through recent history cycle by cycle.
- Post processing performance analysis.
- Automated testing.

8 Summary

The eTPU is a very powerful device, more complicated and with many more dimensions than a CPU. The unique channel design allows the processing engine to meet the drive requirements of practically any physical control system. However, with the power comes a complexity that can be daunting to the new user. This note was intended to suggest some steps for a successful design approach, and to point out possible traps and pitfalls in eTPU systems design. The advice is drawn from experience with real systems. In addition to the design hints, it suggests sources for further information.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-0047, Japan
0120 191014 or +81 3 3440 3569
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2848
Rev. 0
09/2004