Mod5234 eTPU

Application Note

Revision 1.1
September 24, 2007
Document Status: Released

Table of Contents

1 Introduction

This application note will provide a walkthrough for using all of the unique NetBurner eTPU API as well as getting started with any of the eTPU functions that are provided by Freescale. The eTPU (Enhanced Time Processing Unit) is a highly configurable peripheral that is new to the Freescale Coldfire line of microprocessors. The eTPU is a programmable I/O controller with its own core and memory system, allowing it to perform complex timing and I/O management independently of the CPU. The eTPU is essentially an independent microcontroller designed for timing control, I/O handling, serial communications, and motor control applications.

The eTPU on the MOD5234 has 16 channels which are brought out on header pins J2-5 to J2-20. The current NNDK toolset will allow you to easily configure the eTPU channels to perform any function that are available for this peripheral (30+ functions provided by Freescale or custom coded functionality using third party eTPU compilers). The main features provided in the NNDK toolset for the eTPU are: initialization, simplified GPIO with the Pins class, interrupt driven UARTs with file descriptor API, interrupt configuration and NetBurner compatible versions of Freescale's eTPU API.
More detailed documentation describing the eTPU can be found in the MCF5234 user manual and Freescale's eTPU web page:
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=eTPU

Use of the eTPU requires a NetBurner NNDK Tools release of 2.1 rc2 or later.

2 Creating an eTPU Application

Creating an eTPU application involves the following steps. Each of these steps will be covered in detail.

1. Decide what types of eTPU functions you want to implement.
2. Functional Sets. Determine if these functions are part of a predefined “Functional Set” (see the eTPU Functional Set section). If they are, then you can simply include the predefined set. If they are not, then you need to build a custom set using Freescale’s eTPU web application.
3. eTPU Register Configuration. You can use the NetBurner predefined register configuration, or you can create your own register configuration using the Freescale eTPU configuration tool. The NetBurner configuration should cover 99% of all applications.
4. eTPU peripheral initialization. Initialize the eTPU to enable the selected eTPU functions (eg eTPUInit( )).
5. Interrupt or polled mode. If you are using polling mode for your eTPU functions there is nothing else to do. If you want interrupt functionality you will need to create an interrupt service routine and call the appropriate interrupt initialization function.
6. eTPU chanel initialization. Call the appropriate function to enable each of the eTPU channels.
7. The eTPU channel will now operate as described in the Freescale manual.

3 eTPU Source Code Structure

The NNDK includes a large library dedicated to the eTPU for the MOD5234. This section of the application note will give a description of what functionality these files provide to the eTPU. The root directory for the eTPU headers (…\Nburn\MOD5234\include\ETPU) and eTPU C/C++ source (…\Nburn\MOD5234\system\ETPU) includes the following folders and files:

ETPU.h / ETPU.cpp – Contains the functions for all of the post-initialization NetBurner API for the eTPU. The functions are all documented within this header file and also in the NetBurner Runtime Library manual. There is a detailed explanation on how to use the NetBurner eTPU API in the NNDK eTPU API section of this document. These source files were all developed by NetBurner.

ETPUinit.h / ETPUinit.cpp – Contains configurations and function needed for initializing the eTPU. There is a detailed explanation on how to use these files in the eTPU Initialization section of this document. Information on customizing these files can be fopund in the Register Configuration section of this application note. These are modified files that were initially generated with Freescale's eTPU Graphical Configuration Tool.

etpu_struct.h – Contains a struct that maps the internal RAM and registers of the eTPU. This is similar to the NetBurner's sim struct for mapping the processors internal registers. NetBurner applications have no need to access this struct since the NetBurner API in eTPU.h will do it for you. This is a source file provided by Freescale and should not be modified.

mcf523x_vars.h – The addresses of the eTPU RAM and struct are located here. This is a modified source file provided by Freescale and should not be changed.

…\Nburn\MOD5234\include\ETPU\functions_API /
…\Nburn\MOD5234\system\ETPU\functions_API – These directories contains the Freescale API for all eTPU functions they provide. Most of these functions have application notes and examples available for download from the Freescale eTPU product page (http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=eTPU). There is a more information on how to use these files in the Freescale eTPU API section of this document. These files were all provided from Freescale, with some minor modifications made for the MOD5234.

…\Nburn\MOD5234\include\ETPU\sets – This folder contains eTPU function sets that are loaded during eTPU initialization. The function sets contain the binary micro-code for desired functions to use on the eTPU. The NNDK includes the sets provided by Freescale (sets 1, 3, 4) and the also a default NetBurner set (NNDK_etpu_set). There is more information on which functions these sets include and how to create your own

custom set in the Function Sets section of this document. These files were all provided from Freescale, or created with Freescale's eTPU Function Selector web tool and should not be modified.

…\Nburn\MOD5234\include\ETPU\cpu – The files in this directory contain specific configurations for each function included in a set. The included cpu files will work with all four sets provided with the NNDK. There is more information on which functions these sets include and how to create your own custom set in the Function Sets section of this application note. These files were all provided from Freescale, or created with Freescale's eTPU Function Selector web tool and should not be modified.

…\Nburn\MOD5234\include\ETPU\originals – This directory contains the default versions of the NNDK eTPU set and the CPU headers. This is useful if you replace these files and wish to switch back to the default files.

4 eTPU Function Sets

An eTPU "Function Set" is a header file that contains the micro code for the functions you wish to use on the eTPU. These functions must all be part of the same micro-code set to be able to use them simultaneously. When the eTPU is initialized, this micro-code will be loaded into the instruction RAM of the eTPU. Due to a limited size of RAM, it is not possible to have every eTPU function loaded at the same time. The included sets can be found in the following directory: …\Nburn\MOD5234\include\ETPU\sets.

There are 3 options when choosing your Functional Set:

1. Use the set defined by: NNDK_etpu_set.h. If the eTPU functions you want to implement are in this set, then this is the easiest choice. No modifications are needed and you can proceed to the next step.
2. Use one of the other provided predefined sets: etpu_set1.h, etpu_set3.h, or etpu_set4.h. To use one of these sets you need to modify ETPUinit.cpp as described later in this section.
3. Create a custom set using the Freescale Function Selector, as described later in this section.

4.1 Predefined eTPU Functions

The following functions are currently available from Freescale for use on the eTPU:

General Pin Input / Output	(GPIO)
Pulse Width Modulation	(PWM)
Input Capture	(IC)
Output Compare	(OC)
Frequency and Period Measurement	(FPM)
Pulse / Period Accumulation	(PPA)
Queued Output Match	(QOM)
Synchronized Pulse Width Modulation	(SPWM)
Test - helps to test other eTPU functions	(TEST)
Synchronous Peripheral Interface	(SPI)
Universal Asynchronous Reciever / Transmitter	(UART)
UART with Flow Control	(UART_FC)
Stepper Motor	(SM)
Hall Decoder	(HD)
Quadrature Decoder	(QD)
Quadrature Decoder – Home	(QDHOME)
Quadrature Decoder – Index	(QDINDEX)
PWM - Master for DC Motors	(PWMMDC)
PWM - Master for AC Motors	(PWMMAC)
PWM - Full Range	(PWMF)
PWM - Commutation Capable	(PWMC)

Analog Sensing for DC Motors	(ASDC)
Analog Sensing for AC Motors	(ASAC)
Current Controller	(CC)
Speed Controller	(SC)
DC-Bus Brake Controller	(BC)
PMSM Vector Control	(PMSMVC)
ACIM V/Hz Contol	(ACIMVHZ)
ACIM Vector Control	(ACIMVC)

4.2 Predefined Function Sets in the NNDK

The following lists all the available functions in each of the sets provided with the NNDK:
NNDK_etpu_set.h : GPIO, PWM, IC, OC, FPM, PPA, QOM, TEST, SPI, UART, UART_FC
etpu_set1.h: GPIO, PWM, IC, PPA, UART, QOM, SM, SPI, TEST
etpu_set3.h: GPIO, PWMMDC, PWMF, PWMC, HD, QD, QDHOME, QDINDEX, ASDC, CC, SC, BC
etpu_set4.h: GPIO, PWMMAC, PWMF, QD, QDHOME, QDINDEX, ASAC, ACIMVHZ, PMSMVC, ACIMVC, SC, BC

To switch between sets you must change the included set file in ETPUinit.cpp. After the set header file is changed then the MOD5234 system directory must be recompiled to reflect the changes.

4.3 Creating a Custom Function Set

A custom set can be created if none of these sets contain all the functions needed. The Freescale eTPU Function Selector web application (http://www.freescale.com/webapp/etpu/) will allow you to select the various functions you wish to use and also give feedback on how much RAM is available after each function is added. To be able to use the NetBurner API with your custom set you must include the GPIO function. This web application has issues with Firefox but will work with Internet Explorer. When the function selection is completed you will be able to download a package that includes the compiled micro-code for the eTPU. The NNDK already includes most of the source files in this package and only a few will be needed for your custom set. "\etpu_etpu_set\etpu_set.h" is the location of the custom set file in your package. This should be renamed and relocated to the NetBurner system directory "…\Nburn\MOD5234\include\ETPU\sets". Also all of the files in the package directory "\etpu_etpu_set\cpu" should replace the existing files located in "…\Nburn\MOD5234\include\ETPU\cpu". The NNDK contains duplicates of the CPU

files in the directory "…\Nburn\MOD5234\include\ETPU\originals" to allow reverting back to the default sets.

5 Register Configuration

Now that we have selected a Function Set, the next step is to initialize the internal registers for the desired eTPU configuration. The default configuration in the eTPUinit source files has the eTPU timers running from the internal PLL clock at their highest frequency. This yields a speed of 36.864 MHz for TCR1 and 9.216 MHz for TCR2. There are also other configurations for input filtering which can be viewed in ETPUinit.cpp. If you wish to use the default configurations then you can skip to the eTPU Initialization section of this application note.

If a custom configuration for the eTPU is desired then the easiest way to do this is by using the Freescale's eTPU Graphical Configuration Tool which is available for download from Freescales eTPU product page (http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=eTPU). You must first go to the top menu "eTPU/options" and add the following folder for the sets location "C:\Nburn\MOD5234\include\ETPU\", followed by restarting the application. Then configure this application to use the MCF523X at a clock speed of 147456000 Hz in the "Processor" tab. Now the set that you will be using should be chosen from the "Function Set" tab. The "Engine A" tab will allow you to change internal configurations of the eTPU such as timer input sources and filters. The "Engine A Channels" tab is useful if you want example initialization code for any of the peripherals in your set. It will better to configure any channel interrupts with the NetBurner API rather then the "CIE" checkbox on this configuration tab. Once all of the configurations have been chosen, use the menu "file/save" to save the configuration source and header files.

Now we must extract the necessary configurations from the generated source file and replace the existing configurations in the default NNDK eTPU initialization file. This default initialization file is also stored in the "etpu\include\originals" directory also so you can always revert back to the original copy. In the newly created init.c file, copy all the code from the "Global Variables" section to the end of the etpu_config structure. Open the file "…\Nburn\MOD5234\system\ETPU\ETPUinit.cpp" and replace the code from the "Global Variables" section to the end of the etpu_config structure. After ETPUinit.cpp is saved, the MOD5234 system directory must be rebuilt to reflect the changes.

The generated init.c file will also contain the initialization code for the functions set up with the generation tool. For simplicity, this code should be used to initialize the eTPU functions in your project files instead of the NNDK system files.

6 eTPU Initialization

At this point we have a Functional Set, and have decided on a Register Configuration. The eTPU must now be initialized before it can be used. The initialization includes loading the micro-code for the selected functions into eTPU RAM and also setting register configurations, such as the speed of the eTPU timers. The NNDK provides a default set of functions and configurations that are the most commonly used. The previous two sections detailed how to select/create a set of functions and the initialization configurations if required.

6.1 Standard NetBurner eTPU Initialization Function

In the project file containing your initializations (main.cpp), you should include the following header files:

```
#include <pins.h>          // PINS class for GPIO
#include <ETPUinit.h>   // Initialization functions for eTPU
#include <etpu.h>          // NetBurner API for eTPU
```

The standard NetBurner initialization function is **int eTPUInit().** This function will initialize all eTPU settings, RAM and start the timers on the eTPU. Once the eTPU is started, this function will configure all of the channels to be GPIO inputs to put the system in a safe state. Channels can switch between function while the eTPU is running so it is OK to first initialize them all to be high impedance. This function will return 0 if OK or an error code from ETPU.h.

6.2 Manual eTPU Initialization

Th eTPUInit() function is normally the only function that will need to be called to initialize the eTPU. If you are having a problem getting a function to properly run on a channel, or if you do not wish to have the channels initialize to as GPIO, you can call the following two functions in order to initialize the eTPU:

void MOD5234_etpu_init()
This function will initialize the eTPU settings and also load the function micro-code into the eTPU RAM. This function should not be used if eTPUInit() is being called and must be called before MOD5234_etpu_start().

void MOD5234_etpu_start()
This function will clear all the interrupts/requests and start the eTPU. This function should not be used if eTPUInit() is being called and must be called after MOD5234_etpu_init().

Note: The eTPU has a RAM malloc issue that can affect the switching of functions but this should be corrected by Freescale in the near future. If you cannot get a function to run properly on a channel, then use this method to initialize and start the eTPU. The SPI example demonstrated this work around initialization.

7 The NetBurner eTPU API

NetBurner has created a library of functions to simplify the use of the eTPU. Each function has detailed description in the header file, as well as the NNDK runtime libraries documentation. The C++ file "…\Nburn\MOD5234\system\ETPU\ ETPUinit.cpp" and header file "…\Nburn\MOD5234\include\ETPU\ ETPUinit.h" contain the source code for this library.

7.1 eTPU GPIO:

NetBurner provides a simple solution when basic GPIO is needed on the eTPU channels. The PINs class will work on the eTPU pins like all other GPIO pins, but the eTPU must first be initialized. If you are unfamiliar with the NetBurner PINs class, there is an application note available on the NetBurner technical documents webpage.

Here is a sample of what code is needed to toggle pin 10 on J2 (eTPU channel 4) every 2 seconds:

```
#include <pins.h>            //These includes should go after any standard includes
#include <eTPUinit.h>
#include <eTPU.h>

eTPUInit();   //Call once with initialization functions, puts all channels in hiz state

while(1)
{
    J2[10] = 0;
    OSTimeDly( 2 * TICKS_PER_SECOND );
    J2[10] = 1;
    OSTimeDly( 2 * TICKS_PER_SECOND );
}
```

The eTPU also provides advanced GPIO functions such as pin history and interrupts. To access advaced GPIO functionality the Freescale API (…\Nburn\MOD5234\include\ETPU\functions_API\ etpu_gpio.h) should be used. Read the Freescale API section for more details on using these functions.

7.2 eTPU UARTs:

The eTPU with the NetBurner API has the ability to configure any of the channels to function as a UART with a standard C/C++ file descriptor interface. All available functions for a file descriptor are documented under the I/O system chapter of the NetBurner Runtime Libraries documentation. Each channel also has the ability to be independently configured allowing for 8 full duplex UARTs or 16 half duplex UARTs.

int eTPUFDUartOpen(int RXchannel, int TXchannel, int baud_rate = 115200,
int bits_per_data_word = 8,
int parity = FS_ETPU_UART_NO_PARITY,
int priority = FS_ETPU_PRIORITY_MIDDLE);

This function will initialize one or two eTPU channels to have UART functionality. This is an overloaded function, for the most common UART options only the two channel parameters need to be included in the function call. If you need only one UART direction (RX or TX but not both) then the channel parameter for the unused direction should be set to -1. The baud_rate parameter is the speed in bits per second in which the UART will operate. The amount of bits in each serial word can be configured with bits_per_data_word from 1 to 23. The parity parameter should be assigned a value of FS_ETPU_UART_NO_PARITY, FS_ETPU_UART_ODD_PARITY, or FS_ETPU_UART_EVEN_PARITY. This priority parameter should be assigned a value of ETPU_PRIORITY_HIGH, ETPU_PRIORITY_MIDDLE or ETPU_PRIORITY_LOW.

This example code will send “Hello World” on eTPU channel 14 after a character is inputted on channel 15:

```
#include <eTPUinit.h>  //These includes should go after any standard includes
#include <eTPU.h>

eTPUInit();   //Call once with initialization functions, puts all channels in hiz state
int eTPUfd = eTPUFDUartOpen(15, 14 ); // Call once to initialize eTPU UART channels

BYTE temp[20];  // Temp memory used for flushing the UART RX channel
while(1)
{
    if(dataavail(eTPUfd))   // Check if input received on channel 15
    {
        writestring(eTPUfd, “Hello eTPU!”);  // Write the string out channel 14
        read(eTPUfd, temp, 20 );  // Flush the receive channel 15 of input chars
    }
}
```

7.3 Global eTPU Functions:

void fs_timer_stop(void);
This function will disable all of the timers of the eTPU. The eTPU will not process any inputs or micro-code when its timers are disabled. Calling fs_timer_start(); from etpu_util.h will re-start the eTPU timers.

void eTPUCloseChannel(int channel);
This function will disable an ETPU channel, disable any interrupts for that channel, clear event flags for the channel, free the memory allocated if the channel's FIFO was initialized and set all the eTPU struct parameters back to default.

void eTPUInitFIFO(int channel, int MaxSegments = 1);
This is useful if you wish to have storage space dedicated to a single channel of the eTPU. This function will initialize a standard storage FIFO class member from the NetBurner buffers.h library. The FIFO can then be accessed from the channel struct "eTPURec[channel].FIFO" which is declared as an extern in eTPU.h. The MaxSegments parameter is used to set the maximum number of buffers to allocate to the FIFO. Each buffer is 1548 bytes and the FIFO will allocate and de-allocate the buffers as needed, up to the declared MaxSegments. The following functions can be used to manage data for a specific channel with the FIFO:

```
WORD SpaceUsed();                          //Returns the number bytes currently in storage
WORD SpaceAvail();                         //Returns the number of free bytes
int ReadData( PBYTE pCopyTo, int max_bytes );      //Returns the number of bytes read
int WriteData( PBYTE pCopyFrom, int num_bytes ); //Returns number of bytes written
BOOL Empty();                              //Returns true if empty
BOOL Full();                               //Returns true if full
```

The following example code checks how many bytes currently stored FIFO:

```
int channel = etpu_channel_with_FIFO_to_check;
WORD BytesUsed = eTPURec[channel].FIFO.SpaceAvail();
iprintf("There are currently %d bytes stored in the FIFO", BytesUsed);
```

7.4 eTPU Events:

Events are used to communicate between the eTPU and the Coldfire system core. Every function on the eTPU will trigger an event flag to alert the main processor that something has occurred on that channel. If interrupts/DMA is enabled for the channel then this event will also trigger a processor interrupt or DMA request. If neither DMA nor interrupts are being used then polling must be used to determine if anything has changed for a specific channel.

BYTE eTPUEventPoll(int channel, int delay = -1);
This function will poll a specific channel for an event. The "channel" parameter is used to specify which of the 16 channels we are checking for an event. The "delay" parameter will specify how many ticks to poll the channel before timing out. If the "delay" parameter is not included when this function is called then the default value of '-1' will be used and the function will poll until an event occurs. Function returns 0 if event has occurred and 1 if time-out occurred.

void eTPUEventClear(int channel);
This function clears the event flags in the ETPU registers for an ETPU channel. This should be called in an ETPU interrupt routine or after successfully polling an event on a channel. A channel can not have an alert the Coldfire core that an event occurred unless the previous event has been cleared.

7.5 eTPU Interrupts:

This section of the application note will focus on interrupts relating to the eTPU. If the concept of interrupts is new to you then please review the other NetBurner documents and application notes relating to interrupts. Enabling an interrupt for a specific channel will allow the core processor to be immediately notified when an eTPU detects an event occurred on that channel. This is especially useful when performing high speed functions since the Coldfire core will immediately be able to process any incoming or outgoing data.

The following functions were created by NetBurner to simplify the use of interrupts on the eTPU:

void eTPUInterruptEnable(int channel, int level = -1, int priority = -1);
This function should be used instead of the standard NetBurner SetIntc() interrupt function. Calling this function will set all the normal Coldfire interrupt initialization and also enable the interrupts in the ETPU peripheral registers for a specific channel. If no level or priority is included when calling this function then a default value from etpu.h will be assigned. No two interrupts on the Coldfire can have the same level AND priority.

void eTPUInterruptDisable(int channel);
Calling this function will disable the specific ETPU channel interrupts at both the CPU and eTPU register levels.

int (*ETPU_Interrupt_Routine)(int channel);
This callback function points to the interrupt routine that is called when an ETPU interrupt occurs. To create an interrupt routine for the ETPU channels you need to create a function in the form:
int My_ETPU_ISR(int channel);
The callback function should then point the interrupt service routine(ISR) you created:
ETPU_Interrupt_Routine = My_ETPU_ISR;

int (*ETPU_Interrupt_Routine_UART)(int channel);
This callback function works the same way as the previous one except it will allow the interrupt driven UARTs , provided by the NetBurner API, to function properly.

Any time an interrupt occurs on the eTPU, this function will automatically be called and the channel parameter will be passed in. With a conditional statement on the passed in channel parameter, this single ISR can handle interrupts on any of the channels of the eTPU. If you wish to have a function specific ISR, instead of the channel specific method, the Function must be declared for each channel that interrupts. An extern array (one for each channel) of structs is available in from ETPU.h and the Function variable can be accessed like so: 'eTPURec[channel].Function'. More function variables can be added to the current define list at the bottom of ETPU.h if needed. When a specific eTPU channel is initialized or changed, the Function variable for that channel should be set to the proper value. This is how the default ISR (int eTPUDoISR(int channel); in ETPU.cpp) will process interrupts for the UART API.
There is a default interrupt routine in ETPU.cpp (int eTPUDoISR(int channel)) that will be used if no user ISR is declared. The default ISR in ETPU.cpp includes the functions necessary to operate the eTPU UARTs with the NetBurner (ETPU.h) API. Be aware that redirecting the ETPU_Interrupt_Routine pointer will disable the NetBurner interrupt driven UART functions. You can maintain the functionality of the NetBurner UART API in your own ISR by using the ETPU_Interrupt_Routine_UART callback function instead of ETPU_Interrupt_Routine.

8 Freescale eTPU API

All of the functionality that the eTPU can perform is supported with a library of APIs provided by Freescale. A good source of information for the various functions available can be found at Freescale's eTPU product page, which contains many application notes and examples for using their various eTPU functions:
(http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=eTPU)
The Freescale API is the underlying code in most of the NetBurner eTPU API functions. It is recommended to review the GPIO and UART functions in eTPU.cpp be since they demonstrate the proper usage of the Freescale API.

One important concept when using the Freescale API is that most function calls will trigger an event. Make sure all channel events are polled, or have interrupts enabled. Once an event is triggered, it can not be re-triggered until cleared by the user code. The NetBurner eTPUEventClear function can be called to clear events that are polled or trigger interrupts.

Freescale designed their eTPU libraries to run on the Codewarrior tools and NetBurner has modified them to compile for the NetBurner MOD5234 platform. The source and header for all available libraries are included in the NNDK and should not be replaced with the Freescale versions available from their eTPU product page.

9 eTPU Examples

This document was part of a package that contains various examples for the MOD5234 eTPU. Each example is described below:

eTPU_GPIO - This eTPU example and demonstrates how to perform simple GPIO to toggle an eTPU channel.

eTPU_FD_UART - A full duplex UART is created in this example using the NetBurner file descriptor API.

eTPU_15TX_UART – This example configures 15 eTPU channels to be UART transmit only file descriptors.

ETPU_IRQ – This demonstrates how to configure an eTPU channel to have interrupts enabled with a custom ISR. The Freescale GPIO API is used to create inputs that interrupt on a negative edge.

ETPU_PWM - Freescale API is used to enable PWM functionality for each eTPU channel.

ETPU_SPI – Demonstrates how to configure the eTPU SPI to communicate with the Microchip MCP3304 A2D IC. SPI is also one of the Freescale functions affected by the eTPU's micro-code malloc error. The fix for this problem is demonstrated, where the SPI channels can not be initialized as GPIO.