MOTOROLA
intelligence everywhere™

digital dna™

ETPURM/D
5/2004
REV 1

Enhanced Time Processing Unit (eTPU) Preliminary Reference Manual

PRELIMINARY—SUBJECT TO CHANGE WITHOUT NOTICE

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:

Motorola Japan Ltd.; SPS, Technical Information Center
3-20-1, Minami-Azabu Minato-ku, Tokyo 106-8573, Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre, 2 Dai King Street
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://motorola.com/semiconductors/

MOTOROLA

ETPURM/D 5/2004 REV 1

PRELIMINARY—SUBJECT TO CHANGE WITHOUT NOTICE

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Paragraph Number | Title | Page Number

Chapter 1 Enhanced Time Processing Unit (eTPU) Overview

The eTPU is an intelligent, semi-autonomous co-processor designed for timing control. It operates in parallel with the host CPU. The eTPU processes instructions, real-time input events, performs output waveform generation, and accesses shared data without the host CPU's intervention. Consequently, the host CPU setup and service times for each timer event, are minimized or eliminated.

The eTPU has a more powerful processing unit than its predecessors. This more powerful processor allows the eTPU to handle high-level C code very efficiently. A C compiler allows customers to develop customized functions for the eTPU. In addition to a compiler, a high-level assembler and documentation are available for customer development.

The eTPU is an enhanced version of the TPU module. Although there is no compatibility at microcode level, eTPU maintains several features of older TPU versions and is conceptually almost identical to the TPU. These facts, along with a C compiler, make it relatively easy to port older applications, at the same time adding several features listed in Section 1.2.2, "eTPU Enhancements over TPU3."

The eTPU's architecture aims at high resolution/performance timing capabilities. High resolution timing is usually limited by host CPU overhead required to service timing tasks such as period measurement, pulse measurement, pulse width modulated waveform generation, etc. High resolution timing is achieved by three main capabilities on the eTPU:

- Reduced timer function latency, that is the interval from occurrence of an event to the start of event servicing pin actions is immediate.

  The eTPU has dedicated channel hardware that implements essential timer functionality. A time base match generates a pin transition. Capture registers record input transitions.
- Reduced or eliminated host interrupt service time.

  Many interrupts, service requests, are handled by the eTPU microengine, thus freeing the host processor to handle other operations.
- Double action channel capability reducing the channel request rate.

Every eTPU has two match and two capture registers, as opposed to the previous TPUs which only had one of each register. The doubling of these registers allows the generation/capture of complex waveforms with a reduction in required servicing by the eTPU microengine.

The eTPU provides higher resolution than the host CPU can achieve. This is partially due to the eTPU implementation, which includes specific instructions for handling and processing time events. In addition, channel conditions are available for use by the eTPU processor, thus eliminating many branches. The eTPU creates no host overhead for servicing timing events. There are two types of timing events:

- Input pin transition, that is capture.
- Selected time base match, that is, a selected time base counter reached or exceeded a pre-programmed value

Service time is the time spent servicing an event. In general, the service time in microcontrollers is constrained because the instruction set is not optimized for time function synthesis. The eTPU instruction set is optimized for time operations, so that time functions can be implemented with much fewer instructions than the host CPU.

Instructions executed by the eTPU are connected directly to eTPU timing hardware. Knowledge of the hardware conditions allows for faster execution of code by reducing the number of branches, that is parallelism of hardware related actions is enabled by this knowledge of the hardware channel conditions.

1.1 Overview

Figure 1-1 shows a top-level eTPU block diagram. It displays a dual eTPU engine configuration.

NOTE

A single eTPU engine configuration is also possible.

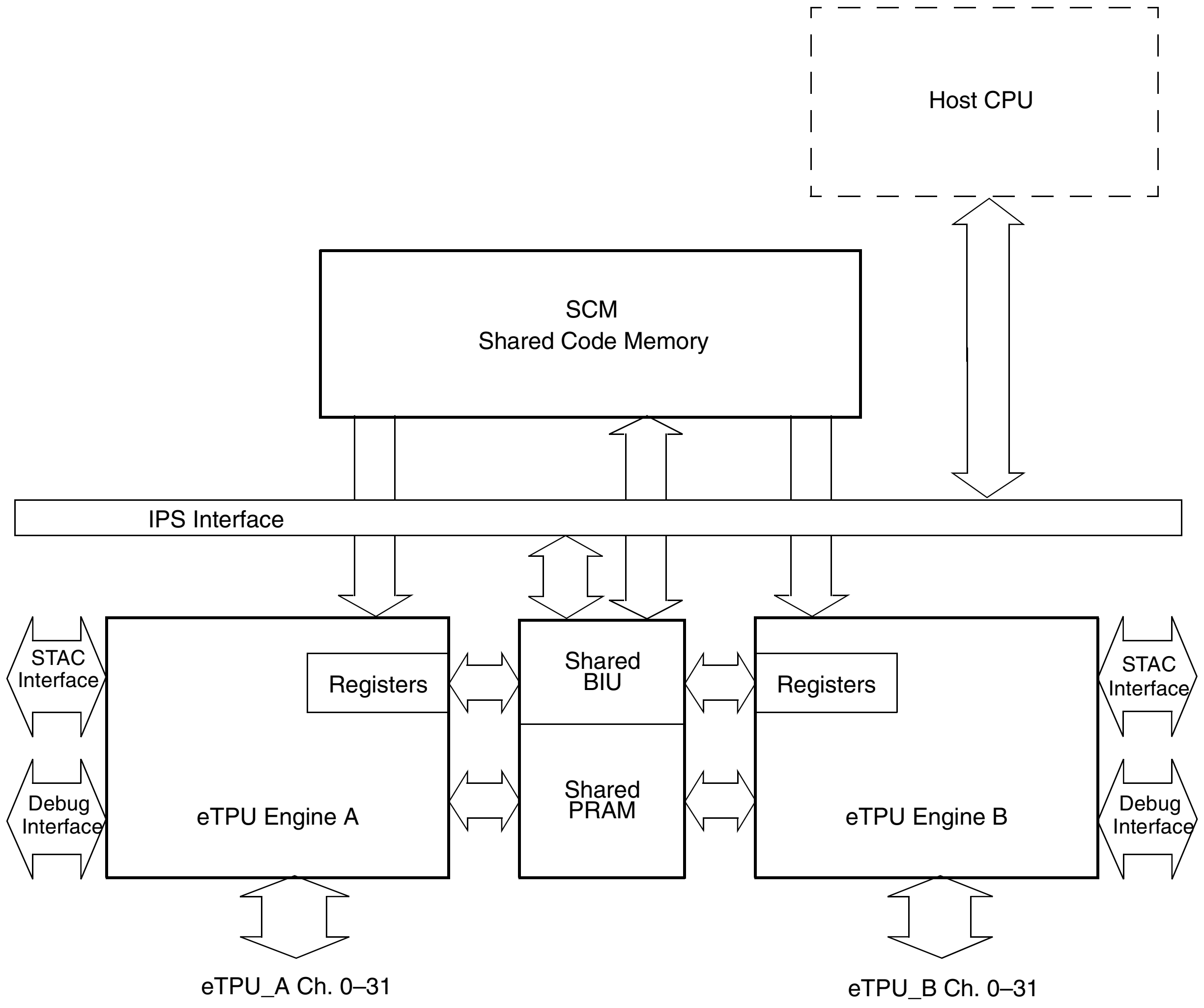


Figure 1-1. eTPU Block Diagram

1.1.1 eTPU Block Components

The eTPU engine is responsible for processing input pin transitions and output pin waveform generation based on time bases. For more information on time bases see Section 1.1.3.1, “Time Bases.” Each eTPU engine has its own microprocessor and dedicated hardware for processing signals on I/O pins. Each eTPU engine also has the ability to interface with external time bases.

Both eTPU engine processors, hereafter called microengines, fetch microinstructions from shared code memory (SCM).

Shared Parameter RAM (SPRAM) holds eTPU application parameters and work data. It is accessed by the host CPU and both microengines.

The bus interface unit (BIU) allows the host CPU to access eTPU registers and data memory.

Each eTPU engine interfaces with 32 I/O channels. Each channel is provided with hardware dedicated for input signal processing and output signal generation. Each channel can also use two shared 24-bit counter registers for its time base.

Each I/O signal pair is associated with a dedicated channel, which provides hardware for input signal processing and output signal generation, in relationship with a selected time base.

1.1.2 eTPU Operation Overview

The eTPU is a real-time microprocessed subsystem: it runs microengine code from instruction memory (SCM) to handle specific events. The eTPU accesses data memory (SPRAM) for parameters, work data and application state info. Events may originate from I/O channels (due to pin transitions and/or time base matches), host CPU requests or inter-channel requests. Events that call for local eTPU processing activate the microengine by issuing a service request. The service request microcode may set an interrupt to the host CPU.

NOTE

I/O channel events cannot directly interrupt the host CPU.

Each channel is associated with a function, which defines its behavior. A function is a software entity consisting of a set of microengine routines, called threads, that attend to eTPU service requests. Function routines are also responsible for channel configuration. Function routines reside in SCM. A function may be assigned to several channels, but a channel can be associated with just one function at a given moment. The eTPU has the capability to change the function assigned to a channel as long as the channel is not currently being serviced. The association between functions and channels is defined by the host CPU, and is explained in detail in Section Chapter 7, “Functions and Threads.”

The eTPU hardware supplies resource sharing features which supports concurrency:

- A hardware scheduler dispatches the service request microengine routines based on a set of priorities defined by the host CPU. Each channel has its own unique priority assignment that primarily depends on CPU assignment. The channel’s number is an inherent property also used to determine priority.
- A service request routine cannot be interrupted until it ends, that is until an end instruction is issued. This sequence of uninterrupted instruction execution is called a thread.

NOTE

A thread can be interrupted only by reset or a forced end (set ETPUECR[FEND] = 1).

- Channel-specific context (registers and flags) are automatically switched between the end of a thread and the beginning of the next one.
- SPRAM arbitration, a dual-parameter coherency controller and semaphores can be used to ensure coherent access to eTPU data shared by both eTPU engines and the host CPU.

1.1.3 eTPU Engine

Each eTPU engine consists of the following blocks: two 24-bit time base count registers, 32 independent timer channels, a task scheduler, a microengine, and a host interface. These blocks are duplicated in a dual eTPU configuration. In addition, a 32-bit Shared Parameter RAM (SPRAM) is used for two eTPU engines data storage and for passing information between the eTPU engines and the host CPU.

Figure 1-2 shows the block diagram for the eTPU engine.

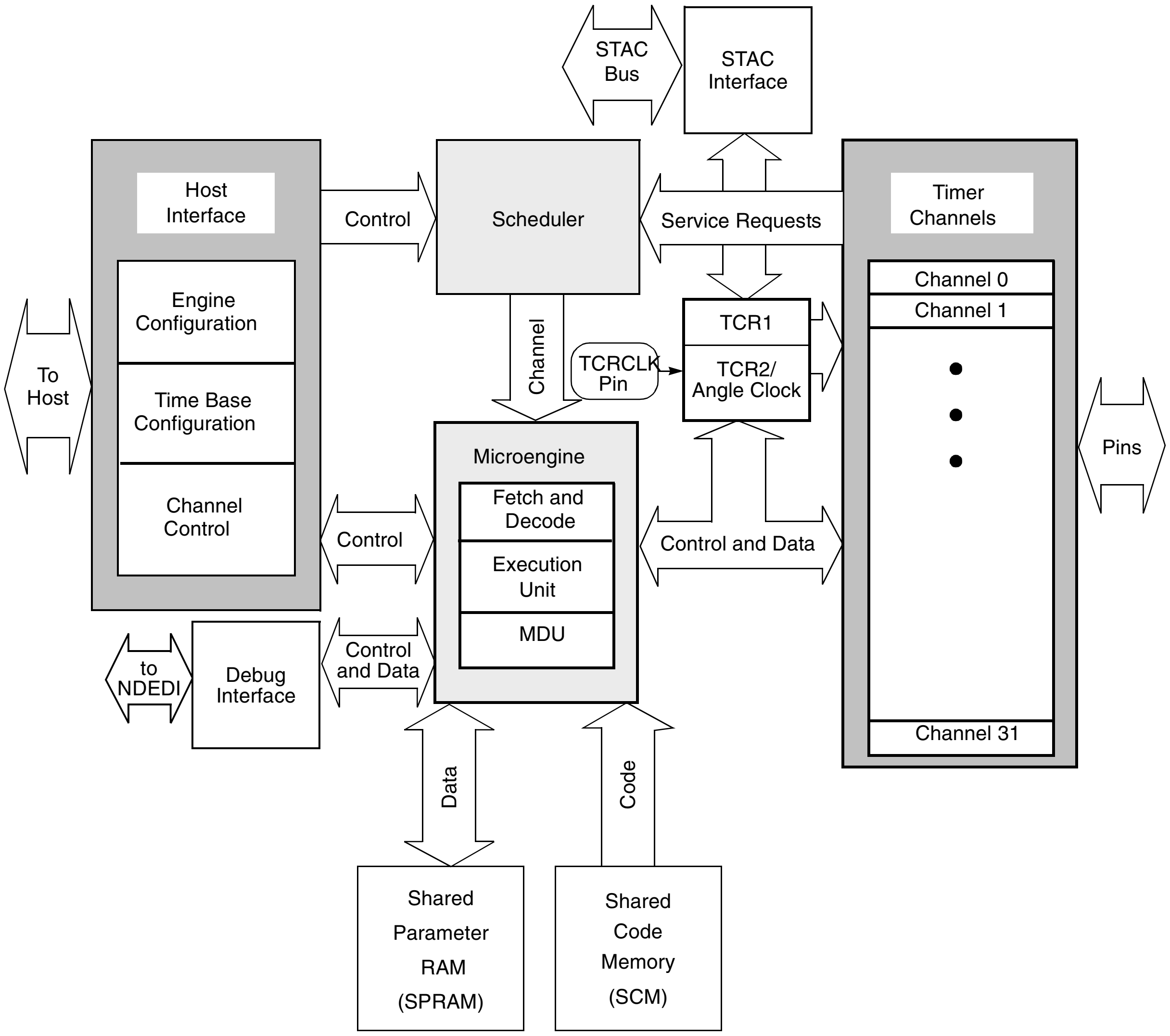


Figure 1-2. eTPU engine Block Diagram

eTPU engines A and B are often referred to as **eTPU A** and **eTPU B** in this document.

1.1.3.1 Time Bases

Each eTPU engine has two 24-bit count registers TCR1 and TCR2 which provide reference time bases for all match and input capture events. Prescalers for both time bases are controlled by the host CPU through bit fields in the eTPU engine configuration registers.

The values for each of TCR1 and TCR2 counter registers can be independently derived from the system clock or from an external input via the TCRCLK pin. In addition, the TCR2 timebase can be derived from special angle-clock hardware which enables implementing angle-based functions. This feature is added to support advanced angle based engine control applications.

For further details refer to Section 5.9, “Time Bases.”

1.1.3.2 eTPU Timer Channels

Each eTPU engine has 32 identical, independent channels. Each channel corresponds to an Input/Output signal pair. Every channel has access to two 24-bit counter registers, TCR1 and TCR2.

Each channel consists of event logic which supports a total of four events, two capture and two match events. The event logic contains two 24-bit capture registers and two 24-bit match registers. The match registers are compared to a selected TCR by greater-than-or-equal-to and equal-only comparators. The match and compare register pairs enable many combinations of single and double-action functions while only requiring a single service from the microengine.

The channel configuration can be changed by the microengine. Each channel can perform double capture, double match or a variety of other capture-match combinations. A channel may be configured so that a match must be recognized on a specified match register before a match event can be recognized on the second match register, that is an ordered match. Some modes are also provided that can block one match by the occurrence of another match, see Table C-1. Service requests may be generated on one or both of the match events.

Digital filters are provided for the input signals, with distinct filtering modes available.

Every channel can use any time base or angle counter for either match or capture operation. For example, a match on TCR1 can capture the value of TCR2. The channels can request service from the microengine due to recognized pin transitions (input events) or timebase matches.

The eTPU channels also support the basic single-action operations found on TPU3 functionality with an increased time resolution of 24 bits, vs. 16 bits on the TPU3.

Every eTPU channel may be configured with the following combinations:

- Single input capture, no match (TPU3 functionality).
- Single input capture with single match time-out (TPU3 functionality).
- Single input capture with double match time-out with several double match sub-modes, see Table C-1.
- Double input capture with single or double match time-out with several double match sub-modes.
- Single output match (TPU3 functionality).
- Double output match with several double match sub-modes.
- Input-dependent output generation.

The double match functionality has various combinations for generation of service request and determining pin actions. For more details refer to Section 5.8, “Enhanced Channels.”

1.1.3.3 Host Interface

The host interface allows the host CPU to control the operation of the eTPU. In order for the eTPU to start operation, the host CPU must initialize the eTPU by writing to the appropriate host interface registers to assign a function and priority to each channel. In addition, the host writes to the host service request and channel configuration registers to further define operation for each initialized channel. Refer to Chapter 5, “Host Interface,” for a detailed description.

NOTE

The host must first initialize the memory prior to enabling any eTPU function. Then the host enables eTPU access to the SCM (which also disables host access).

1.1.3.4 Shared Parameter RAM (SPRAM)

The SPRAM works like data RAM which can be accessed by the host CPU and up to two eTPU engines. This memory is used for either:

- Information transfer between the host CPU and the eTPU.
- As data storage for the eTPU microcode program.
- For communication between the two eTPU engines.

The SPRAM width is 32 bits, and is accessible by the host in any of the three formats: byte, 16-bit, or 32-bit. The eTPU can access the SPRAM’s full 32 bits, lower 24 bits or upper byte (8-bit).

The host can also access the SPRAM space mirrored in other areas with parameter sign extension (PSE). PSE allows for data with fewer than 32 bits in another address area to be accessed as 32 bit sign-extended data without using the host’s bandwidth to extend the data. Parameter signal extension accesses differ from the usual host accesses to the original SPRAM area as follows:

- Writes are effective only to the lower 3 bytes of a word: the word’s most significant byte (byte address) is kept unaltered in SPRAM.

NOTE

For the most significant byte, it should be recalled the word format is big endian, as in the default PowerPC word format.

- Reads return the lower 3 bytes of a word sign-extended to 32 bits, that is: the most significant bit of the words 2nd most significant byte (byte addresses) is copied in all 8 bits of the most significant read byte.

Each eTPU channel can be associated with a variable number of parameters located in the SPRAM, according to its selected function. In addition, the SPRAM can be fully shared between two eTPU engines, enabling communication between them.

High flexibility of the SPRAM utilization is achieved as follows:

- Each channel has a programmable base address pointing to the address of its first parameter with two parameter granularity, that is the base address pointer has a resolution of 2*n, where n is an individual parameter address. This way the SPRAM can be partitioned according to the actual function needs.
- The microcode can access the first 128 parameters of the selected channel in channel relative access mode. The relative address is an offset from the programmable base address mentioned above.
- Each engine can access all the SPRAM address space in indirect addressing mode. Blocks of data are easily transferred using stack operation.
- Absolute addressing mode can access the first 256 parameters (TPU3 functionality), implementing a shared pool of parameters holding global variables.

In the host address space each parameter occupies four bytes (32 bits). eTPU usage of the upper byte is achieved by having a 32-bit P register which can access the upper byte, the lower 24 bits or all the 32 bits. The microcode can switch between access sizes at any time.

Each function may require a different number of parameters. During the eTPU initialization the host has to program channel base addresses, allocating proper parameters for each channel according to its selected function.

1.1.3.5 Scheduler

Out of reset, all channels are disabled. The host CPU makes a channel active by assigning it one of three priorities: high, middle, or low. The scheduler determines the order in which channels are serviced based on channel number and assigned priority. The priority mechanism, implemented in hardware, ensures that all requesting channels are serviced. For additional details refer to Chapter 6, “Scheduler.”

1.1.3.6 Microengine

The eTPU microengine is a simple RISC implementation which performs each instruction in a microcycle of two system clocks, while pre-fetching the next instruction through an instruction pipeline. Instruction execution time is constant for the arithmetic logic unit (ALU) unless it gets wait states from SPRAM arbitration. Two eTPU engines share code memory without having any performance degradation by interleaving their accesses, that is both accesses happen on same microcycle. One engine lags the other by 1/2 a microcycle, but the channels are synchronized to the microcycle of it’s own engine.

The instruction width is 32 bits. The microengine instruction set provides basic arithmetic and logic operations, flow control (jumps and subroutine calls), SPRAM access, and channel configuration and control. The instruction formats are defined in such a way that allow particular combinations of several of these operations with unconflicting resources to be executed in parallel in the same microcycle, thus improving performance.

The microengine has also an independent multiply/divide/MAC unit that performs these complex operations in parallel with other microengine instructions.

Channel functionality is integrated to the instruction set through channel control operations and conditional branch operations, which support jumps/calls on channel-specific conditions. This allows quick and terse channel configuration and control code, contributing to reduced service time.

A detailed description of the microengine is found in Chapter 8, “Microengine.”

1.1.3.7 Dual eTPU engine System

A typical eTPU implementation includes two eTPU engines sharing SPRAM and the same code in SCM.

The two eTPU engines share the bus interface unit (BIU) and the parameter RAM (SPRAM) which enable host CPU to eTPU and eTPU engine to engine communication. The shared BIU includes coherency logic which supports dual parameter (8 bytes) coherency in transfers between the host and eTPU, using a temporary parameter area within the SPRAM. More details on this can be found in Section 5.7, “Parameter Sharing and Coherency.”

1.2 Features

1.2.1 eTPU Feature Summary

The eTPU includes these distinctive features:

- Up to 32 channels for each eTPU engine, each channel is associated with an Input/Output signal pair.
  - Enhanced input digital filters on the input pins for improved noise immunity. The eTPU digital filter can use 2 samples, 3 samples or work in continuous mode.
  - Orthogonal channels, except for channel 0: each channel can perform any time function. Each time function can be assigned to more than one channel at a given time, so each signal can have any functionality. Channel 0 has the same capabilities of the others, but can also work with special Angle Counter logic (see below).

- A link service request allows activation of a channel function by request of another channel, even between eTPU engines.
- A host service request allows activation of a channel function by the host CPU request.
- Each channel has an event mechanism which supports single and double action functionality in various combinations. It includes two 24-bit capture registers, two 24-bit match registers, 24-bit greater-equal or equal-only comparators.

• Two independent 24-bit time bases for channel synchronization:

- The first time base may be clocked by the system clock with programmable prescaler division from 2 to 512 (in steps of 2), or by the output of the second time base prescaler.
- The first time base can also be clocked by an external signal with programmable prescaler division of 1 to 256.
- The second time base may be clocked by an external signal with programmable prescaler division from 1 to 64 or by the system clock divided by 8.
- Both time bases can be exported or imported from engine to engine through the STAC (Shared Time and Counter) bus.

NOTE

An engine cannot export/import to/from itself. An engine cannot import a time base and/or angle count if it is in angle mode.

- The second time base counter can work as an angle counter, enabling angle based applications to match angle instead of time.
- The second time base can alternatively be used as a pulse accumulator gated by an external signal.

• Event-triggered RISC processor (microengine):

- 2 stage pipeline implementation (fetch and execution), with separate instruction memory (SCM) and data memory (SPRAM).
- Two system clock microcycle fixed-length instruction execution for the ALU..
- Interleaved SCM access in dual eTPU engine avoids contention in time for instruction memory.
- Up to 64 kbytes of Shared Code Memory (SCM).
- Up to 8 kbytes of Shared Parameter (data) RAM (SPRAM) with interleaved access in dual eTPU engine avoids contention for data memory.
- Instruction set with embedded channel support, including specialized channel control subinstructions and conditional branching on channel-specific flags.

  - Channel-oriented addressing: channel-bound address mode with host configured channel base address allows the same function to operate independently on different channels. For example, a common spark function can operate with different parameters and different channels.
  - Channel-bound data address space of up to 128 32-bit parameters (512 bytes).
  - Global parameter address mode allows access to common channel data of up to 256 32-bit parameters (1024 bytes).
  - Support for indirect and stacked data access schemes.
  - Parallel execution of: data access, ALU, channel control and flow control subinstructions in selected combinations.
  - 32-bit microengine registers and 24-bit resolution ALU, with 1 microcycle addition and subtraction, absolute value, bitwise logical operations on 24-bit, 16-bit, or byte operands; single bit manipulation, shift operations, sign extension and conditional execution.
  - Additional 24-bit multiply/MAC/divide unit which supports all signed/unsigned multiply/MAC combinations, and unsigned 24-bit divide. The MAC/divide unit works in parallel with the regular microcode commands.
- Resource sharing features resolves channel contention for common use of channel registers, memory and microengine time:
  - Hardware scheduler works as a “task management” unit, dispatching event service routines by pre-defined, host-configured priority.
  - Automatic channel context switch when a “task switch” occurs, that is, one function thread ends and another begins to service a request from other channel: channel-specific registers, flags and parameter base address are automatically loaded for the next serviced channel.
  - Individual channel priority setting in 3 levels: high, middle and low.
  - Scheduler priority scheme allows calculation of worst case latency for event servicing and ensures servicing of all channels by preventing permanent blockage.
  - SPRAM shared between host CPU and both eTPU engines, supporting channel-channel or host-channel communication.
  - Hardware implementation of 4 semaphores allows for resource arbitration between channels in both eTPU engines.
  - Hardware semaphores directly supported by the microengine instruction set.
  - Dual parameter coherency hardware support allows coherent (to host) access to 2 parameters by microengine(s) in back-to-back accesses.
  - Coherent dual-parameter controller allows coherent (to microengines) accesses to 2 parameters by the host.

- Test and development support features:
  - Nexus level 3 debug support through the eTPU Nexus block (NDEDI).
  - Software breakpoints.
  - Debug interface supporting single-step execution, forced microinstruction execution, Hardware breakpoints and watchpoints on several conditions.
  - SCM (code memory) continuous signature-check built-in code integrity test multiple input signature calculator (MISC), runs concurrently with eTPU normal operation.

1.2.2 eTPU Enhancements over TPU3

The eTPU has several enhancements over the TPU3,which are highlighted in the following list:

- 32 orthogonal channels with enhanced functionality. Full support for double action with double match and double transition sub-mode combinations.
- Input and output features separated in channel logic and microinstructions, allowing input and output signals to be processed separately or combined.
- Increased time resolution and execution unit to 24 bits.
- Increased linear code memory, shared by two eTPU engines, configurable up to 16K positions (64 Kbytes).
- Increased parameter RAM address range (16 Kbytes each engine) and width (32 bits per parameter). The parameter RAM can be dynamically allocated to support variable number of parameters for each channel. Each channel can have access to at least 256 parameters.
- The parameter RAM is fully shared by two eTPU engines (SPRAM), supporting inter-engine communication.
- Hardware semaphores to guarantee coherency of multiple-word data from SPRAM.
- Enhanced arithmetic operations, including add/subtract with carry, absolute value, multiple shift and rotate, conditional execution with variable operand widths.
- Enhanced logic operations, including bitwise operations (AND, OR, XOR) and bit manipulation, with conditional execution. Support for read-modify-write of any bit in the SPRAM.
- Hardware for multiply/MAC/divide, running in parallel to execution of other operations. The 24-bit divide result is available after 13 other unrelated instructions. Multiplication supports any data width of both operands (8, 16 or 24 bits), signed or unsigned. A 24x24 multiply/MAC result is available after four other unrelated instructions. A 24x8 multiply/MAC result is available after one other unrelated instruction.

- Supports export/import of internal time bases between eTPU engines or to the eMIOS.
- Contains angle clock hardware, supported by microcode, which can provide up to a 24-bit angle bus instead of time bus. This feature enables the eTPU to run angle based engine control applications.
- More interrupt types. Each eTPU channel can generate a data transfer request interrupt, in addition to regular interrupts, and one global exception interrupt. Data transfer requests are used as DMA requests. This feature takes advantage of DMA peripherals which off-loads the host. Interrupt overflow status is also provided.
- Improved visibility to the host (pin states, time bases, serviced channel).
- An edge case of priority inversion on TPU3 scheduler was resolved.
- Supports channel link requests between eTPU engines.

1.3 Modes of Operation

The eTPU is capable of working in the following modes:

- User configuration mode
  - By having access to the shared code memory (SCM), the CPU has the ability to program the eTPU cores with time functions.
- User mode
  - The CPU does not access the eTPU shared code memory.
  - Use of pre-defined eTPU functions.
- Debug mode

  The CPU debugs eTPU code, accessing special Trace/Debug features via Nexus interface:
  - hardware breakpoint/watchpoint setting
  - access to internal registers
  - single-step execution
  - forced instruction execution
  - software breakpoint insertion and removal.
- Module Disable Mode

  eTPU engine clocks are stopped through a register write to ETPUECR bit MDIS, saving power. Input sampling stops. eTPU engines can be in stop mode independently. Module disable mode stops only the engine clock, so that the shared BIU, and global channel registers can be accessed, and interrupts and DMA requests can be cleared and enabled/disabled. An engine only enters module disable mode when any currently running thread is finished. For more information on thread behavior, see Section Chapter 7, “Functions and Threads.”

- System Stop Mode

  Stop Mode is entered when the eTPU responds to a system asserted stop signal. The definition of which clocks are stopped is made at the MCU level, which defines whether or not registers can be accessed, interrupts and DMA requests cleared.

These modes are loosely selected: there is no unique register field or signals to choose between them. Some features of one mode can be used with features of other mode(s).

1.3.1 eTPU Mode Selection

User and user configuration are the production operating modes, and differ from each other only in access to SCM. User programmability is only possible with RAM SCM.

On chips where eTPU SCM is implemented as a RAM, the SCM can either be accessed directly from IP-Bus for code loading, or for software breakpoint setting. On chips with ROM SCM, an internal SCM Emulation RAM may be used, depending on the specific MCU implementation, to replace ROM SCM for test or debug purposes. SCM Emulation RAM is selected in a MCU-specific way.

For more information on SCM access, Debug and Test features, refer to Chapter 10, “Test and Development Support.”

Module disable mode is entered by setting ETPUECR register bit MDIS. eTPU engines can be individually stopped (there is one ETPUECR register for each engine).

Chapter 2
External Signal Description

2.1 Introduction

There are 69 external signals associated with each eTPU engine: 32 channel input signals, 32 channel output signals, and a TCRCLK clock input, totalling 138 in a Dual Engine system.

Depending on the MCU integration, the input and output signals of a channel can be tied to one pin. In this case, the direction of each channel signal, either output or input, is determined by the activation of an output enable driver signal.

2.2 eTPU Signals

2.2.1 Output and Input Channel Signals

The channel signal connections for eTPU engine A and eTPU engine B are described in Table 2-1 and Table 2-2, respectively. Each eTPU channel has an input and output associated with it. The eTPU microcode may be programmed to set the output level of an eTPU channel in one of two manners:

- By forcing the logic level to a specified value.
- By specifying the logic level output action when a match or transition event occurs.

NOTE

Each eTPU engine has four output disable signals which allow the channel output signal to be forced to a logic level independently of the output value from the channel logic.

Depending on MCU integration, the output signal driver may be enabled by the output buffer enable internal signal which comes from the eTPU. In this case, the output buffer may be controlled by microcode through a specific microinstruction field. There is one independent output buffer Enable signal for each channel. For more information on output control from microcode, refer to section 4.9.3.3 Transition Detection and Pin Action Control.

Every eTPU channel input has a digital filter. This filter is designed to filter out noise pulses that have width less than a specified value. This prevents small noise glitches from being recognized by the transition detect logic. Any pulses wider than the specified filter width will be passed to the channel transition detect logic. For more details on channel input filters, refer to Section 5.8.6, "Enhanced Digital Filter (EDF)."

Table 2-1. eTPU_A Channel Connection Table

eTPU A Signal	Input/ Output	Pin Connections		Signals Pin is Shared With	DSPI Serial Channel Connections
		eTPU Ch	Pin Number		
eTPU_A[0:9]	IN	0 1–4 5–8 9	N3 M4–M1[1] L4–L1[1] K4	eTPU_A[12:21] (output only) GPIO[114:123]	not connected
	OUT				DSPI_C[4:13]
		0–9	AF15, AE15, AC16, AD15, AF16, AE16, AD16, AF17, AC17, AE17	eMIOS[0:9] GPIO[179:188]	
eTPU_A[10:11]	IN	10–11	K3–K2[1]	eTPU_A[22:23] (output only) GPIO[124:125]	not connected
	OUT				DSPI_C[14:15]
eTPU_A[12:15]	IN	12 13–15	K1 J4–J2[1]	GPIO[126:129]	not connected
	OUT				DSPI_C[0:3]
		12 13–15	N3 M4–M2[1]	eTPU_A[0:3] GPIO[114:117]	
eTPU_A[16:19]	IN	16 17–19	J1 H4–H2[1]	GPIO[130:133]	not connected
	OUT				DSPI_B[7:4][1] DSPI_D[5:2][1]
		16 17–19	M1 L4–L2[1]	eTPU_A[4:7] GPIO[118:121]	
eTPU_A[20:21]	IN	20 21	H1 G4	IRQ[8:9] (input only) GPIO[134:135]	not connected
	OUT				DSPI_B[3:2][1] DSPI_D[1:0][1]
		20 21	L1 K4	eTPU_A[8:9] GPIO[122:123]	
eTPU_A[22:23]	IN	22 23	G2 G1	IRQ[10:11] (input only) GPIO[136:137]	not connected
	OUT				
		22 23	K3 K2	eTPU_A[10:11] GPIO[124:125]	
eTPU_A[24:27]	IN	not connected		not connected	DSPI_B[13:10][1]
eTPU_A[24:27]	OUT	24, 25 26, 27	F1, G3 F3, F2	IRQ[12:15] (input only) GPIO[138:141]	DSPI_B[13:10][1] DSPI_D[15:12][1]
eTPU_A[28:29]	IN	not connected		not connected	DSPI_B[9:8][1]
	OUT	28 29	E1 E2	GPIO[142:143]	DSPI_B[9:8][1] DSPI_D[11:10][1]

Table 2-1. eTPU_A Channel Connection Table (continued)

eTPU A Signal	Input/ Output	Pin Connections		Signals Pin is Shared With	DSPI Serial Channel Connections
		eTPU Ch	Pin Number		
eTPU_A[30:31]	IN	30 31	D1 D2	GPIO[144:145]	not connected
	OUT				

1. The channel numbers for some of the DSPI channels connections are reversed, e.g. if eTPU_A[16:19] is mapped to DSPI_B[7:4], then eTPU_A[16] is connected to DSPI_B[7], eTPU_A[17] is connected to DSPI_B[6],..., and eTPU_A[19] is connected to DSPI_B[4]

Table 2-2. eTPU_B Channel Connection Table

eTPU B Signal	Input/ Output	Pin Connections		Signals Pin is Shared With	DSPI Serial Channel Connections
		eTPU Ch.	Pin Number		
eTPU_B[0:7]	IN	0–7	M25, M24, L26, L25, L24, K26, L23, K25	eTPU_B[16:23] (output only) GPIO[147:154]	not connected
	OUT				DSPI_A[15:8][1]
		0–7	AE19, AD19, AF20, AE20, AR21, AC19, AD20, AE21	eMIOS[16:23] GPIO[195:202]	
eTPU_B[8:15]	IN	8–15	K24, J26, K23, J25, J24, H26, H25, G26	eTPU_B[24:31] (output only) GPIO[155:162]	not connected
	OUT				DSPI_A[7:0][1]
eTPU_B[16:31]	IN	16–31	D16, D17, A17, C16, A18, B17, C17, D18, A19, B18, C18, A20, B19, D19, C19, B20	GPIO[163:178]	not connected
	OUT				

1. The channel numbers for some of the DSPI channels connections are reversed, e.g. if eTPU_B[0:7] is mapped to DSPI_A[15:8], then eTPU_B[0] is connected to DSPI_A[15], eTPU_B[1] is connected to DSPI_A[14],..., eTPU_B[7] is connected to DSPI_A[8].

2.2.2 TCRCLK_[A:B], Time Base Clock Signal (TCRCLK)

The TCRCLK_[A:B] input signals are used control the TCR1 and TCR2 time bases for eTPU_A and eTPU_B.

NOTE

Throughout this document, TCRCLK_A and TCRCLK_B are referred to generically as TCRCLK.

There is one independent TCRCLK input for each engine. Table 2-3 shows the TCRCLK pin connections. For pulse accumulator operations TCRCLK can be used as a gate for a counter based on the system clock divided by eight. For angle operations TCRCLK can be used to get the tooth transition indications in angle mode. Further details can be found in Section 5.9, “Time Bases,” and Section 5.10, “eTPU Angle Counter (EAC).”

2.2.3 Channel Output Disable Signals

Each eTPU engine has 4 input signals that are used to force the outputs of a group of 8 channels to an inactive level. These signals originate from the eMIOS. When an output disable signal is active, all the 8 channels assigned to the disable signal that have their ODIS bits set to 1 in ETPUCxCR register have their outputs forced to the opposite of the value specified in the ETPUCxCR[OPOL] bit. For more information on the ETPUCxCR registers see Section 4.6.1, "eTPU Channel x Configuration Register (ETPUCxCR)." Therefore, individual channels can be selected to be affected by the output disable signals, as well as their disabling forced polarity.

The output disable channel groups are defined in Table 2-3.

Table 2-3. Output Disable Channel Groups

eMIOS Channel	Engine	eTPU Channels Disabled
11	A	0 to 7
10		8 to 15
9		16 to 23
8		24 to 31
21	B	0 to 7
20		8 to 15
19		16 to 23
18		24 to 31

Chapter 3 Memory Map

3.1 Introduction

The guideline for the description of all bits and fields throughout Chapter 3, "Memory Map," is to provide only a brief explanation (without examples or method of use, of the features) since it will be used mainly as a reference for the reader that is studying Chapter 5, "eTPU Functional Description," where features are explained in detail.

3.2 Memory Map

The eTPU system simplified memory map is shown in Table 3-1. The base address for the eTPU module is listed as eTPU_BASE. Each of the register areas shown may have their own reserved address areas.

Table 3-2 shows a detailed memory map with where eTPU_BASE is the base address for the eTPU module.

Table 3-1. High-Level Memory Map

Address	Use
eTPU_BASE – eTPU_BASE+0x1F	eTPU System Module Configuration Registers
eTPU_BASE+0x20 – eTPU_BASE+0x2F	eTPU A Time Base Registers
eTPU_BASE+0x30 – eTPU_BASE+0x3F	Reserved
eTPU_BASE+0x40 – eTPU_BASE+0x4F	eTPU B Time Base Registers
eTPU_BASE+0x50 – eTPU_BASE+0x1FF	Reserved
eTPU_BASE+0x200 – eTPU_BASE+0x2FF	eTPU[A:B] Global Channel Registers
eTPU_BASE+0x300 – eTPU_BASE+0x3FF	Reserved
eTPU_BASE+0x400 – eTPU_BASE+0x7FF	eTPU A Channel Registers
eTPU_BASE+0x800 – eTPU_BASE+0xBFF	eTPU B Channel Registers
eTPU_BASE+0xC00 – eTPU_BASE+0x7FFF	Reserved
eTPU_BASE+0x8000 – eTPU_BASE+0x8BFF	SPRAM (3 Kbytes)
eTPU_BASE+0x8C00 – eTPU_BASE+0xBFFF	Reserved

Table 3-1. High-Level Memory Map (continued)

Address	Use
eTPU_BASE+0xC000 – eTPU_BASE+0xCBFF[1]	SPRAM PSE mirror [1] (3 kBytes)
eTPU_BASE+0xCC00 – eTPU_BASE+0xFFFF	Reserved
eTPU_BASE+0x1_0000 – eTPU_BASE+0x1_2FFF	SCM (16 Kbytes)
eTPU_BASE+0x1_3000 – eTPU_BASE+0x1_FFFF	Not writable Value returned determined by ETPUSCMOFFDATAR

1. Parameter Sign Extension access area, see Section 5.2.3, "Parameter Access."

Table 3-2. Detailed Memory Map

Address	Use
eTPU_BASE	eTPU Module Configuration Register (ETPUMCR)
eTPU_BASE+0x04	eTPU Coherent Dual-Parameter Controller Register (ETPUCDCR)
eTPU_BASE+0x08	Reserved
eTPU_BASE+0x0C	eTPU MISC Compare Register (ETPUMISCCMPR)
eTPU_BASE+0x10	Reserved
eTPU_BASE+0x14	eTPU SCM Off Data Register (ETPUSCMOFFDATAR)
eTPU_BASE+0x18	eTPU B Engine Configuration Register (ETPUECR_B)
eTPU_BASE+0x1C	Reserved
eTPU_BASE+0x20	eTPU A Time Base Configuration Register (ETPUTBCR_A)
eTPU_BASE+0x24	eTPU A Time Base 1 (ETPUTB1R_A)
eTPU_BASE+0x28	eTPU A Time Base 2 (ETPUTB2R_A)
eTPU_BASE+0x2C	eTPU A STAC Bus Interface Configuration Register (ETPUREDCR_A)
eTPU_BASE+0x30 – eTPU_BASE+0x3F	Reserved
eTPU_BASE+0x40	eTPU B Time Base Configuration Register (ETPUTBCR_B)
eTPU_BASE+0x44	eTPU B Time Base 1 (ETPUTB1R_B)
eTPU_BASE+0x48	eTPU B Time Base 2 (ETPUTB2R_B)
eTPU_BASE+0x4C	eTPU B STAC Bus Interface Configuration Register (ETPUREDCR_B)
eTPU_BASE+0x50 – eTPU_BASE+0x1FF	Reserved
eTPU_BASE+0x200	eTPU A Channel Interrupt Status Register (ETPUCISR_A)
eTPU_BASE+0x204	eTPU B Channel Interrupt Status Register (ETPUCISR_B)
eTPU_BASE+0x208	Reserved
eTPU_BASE+0x20C	Reserved
eTPU_BASE+0x210	eTPU A Channel Data Transfer Request Status Register (ETPUCDTRSR_A)
eTPU_BASE+0x214	eTPU B Channel Data Transfer Request Status Register (ETPUCDTRSR_B)
eTPU_BASE+0x218	Reserved

Table 3-2. Detailed Memory Map

Address	Use
eTPU_BASE+0x21C	Reserved
eTPU_BASE+0x220	eTPU A Channel Interrupt Overflow Status Register (ETPUCIOSR_A)
eTPU_BASE+0x224	eTPU B Channel Interrupt Overflow Status Register (ETPUCIOSR_B)
eTPU_BASE+0x228	Reserved
eTPU_BASE+0x22C	Reserved
eTPU_BASE+0x230	eTPU A Channel Data Transfer Request Overflow Status Register (ETPUCDTROSR_A)
eTPU_BASE+0x234	eTPU B Channel Data Transfer Request Overflow Status Register (ETPUCDTROSR_B)
eTPU_BASE+0x238	Reserved
eTPU_BASE+0x23C	Reserved
eTPU_BASE+0x240	eTPU A Channel Interrupt Enable Register (ETPUCIER_A)
eTPU_BASE+0x244	eTPU B Channel Interrupt Enable Register (ETPUCIER_B)
eTPU_BASE+0x248	Reserved
eTPU_BASE+0x24C	Reserved
eTPU_BASE+0x250	eTPU A Channel Data Transfer Request Enable Register (ETPUCDTRER_A)
eTPU_BASE+0x254	eTPU B Channel Data Transfer Request Enable Register (ETPUCDTRER_B)
eTPU_BASE+0x258 – eTPU_BASE+0x27F	Reserved
eTPU_BASE+0x280	eTPU A Channel Pending Service Status Register (ETPUCPSSR_A)
eTPU_BASE+0x284	eTPU B Channel Pending Service Status Register (ETPUCPSSR_B)
eTPU_BASE+0x288	Reserved
eTPU_BASE+0x28C	Reserved
eTPU_BASE+0x290	eTPU A Channel Service Status Register (ETPUCSSR_A)
eTPU_BASE+0x294	eTPU B Channel Service Status Register (ETPUCSSR_B)
eTPU_BASE+0x298 – eTPU_BASE+0x3FF	Reserved
eTPU_BASE+0x400	eTPU A Channel 0 Configuration Register (ETPUC0CR_A)
eTPU_BASE+0x404	eTPU A Channel 0 Status and Control Register (ETPUC0SCR_A)
eTPU_BASE+0x408	eTPU A Channel 0 Host Service Request Register (ETPUC0HSRR_A)
eTPU_BASE+0x40C	Reserved
eTPU_BASE+0x410	eTPU A Channel 1 Configuration Register (ETPUC1CR_A)
eTPU_BASE+0x414	eTPU A Channel 1 Status and Control Register (ETPUC1SCR_A)
eTPU_BASE+0x418	eTPU A Channel 1 Host Service Request Register (ETPUC1HSRR_A)
eTPU_BASE+0x41C	Reserved

Table 3-2. Detailed Memory Map

Address	Use
	. . .
eTPU_BASE+0x5F0	eTPU A Channel 31 Configuration Register (ETPUC31CR_A)
eTPU_BASE+0x5F4	eTPU A Channel 31 Status and Control Register (ETPUC31SCR_A)
eTPU_BASE+0x5F8	eTPU A Channel 31 Host Service Request Register (ETPUC31HSRR_A)
eTPU_BASE+0x5FC – eTPU_BASE+0x7FF	Reserved
eTPU_BASE+0x800	eTPU B Channel 0 Configuration Register (ETPUC0CR_B)
eTPU_BASE+0x804	eTPU B Channel 0 Status and Control Register (ETPUC0SCR_B)
eTPU_BASE+0x808	eTPU B Channel 0 Host Service Request Register (ETPUC0HSRR_B)
eTPU_BASE+0x80C	Reserved
eTPU_BASE+0x810	eTPU B Channel 1 Configuration Register (ETPUC1CR_B)
eTPU_BASE+0x814	eTPU B Channel 1 Status and Control Register (ETPUC1SCR_B)
eTPU_BASE+0x818	eTPU B Channel 1 Host Service Request Register (ETPUC1HSRR_B)
eTPU_BASE+0x81C	Reserved
	. . .
eTPU_BASE+0x9F0	eTPU B Channel 31 Configuration Register (ETPUC31CR_B)
eTPU_BASE+0x9F4	eTPU B Channel 31 Status and Control Register (ETPUC31SCR_B)
eTPU_BASE+0x9F8	eTPU B Channel 31 Host Service Request Register (ETPUC31HSRR_B)
eTPU_BASE+0x9FC – eTPU_BASE+0x7FFF	Reserved
eTPU_BASE+0x8000 – eTPU_BASE+0x8BFF	3 kBytes Shared Parameter RAM (SPRAM)
eTPU_BASE+0x8C00 – eTPU_BASE+0xBFFF	Reserved
eTPU_BASE+0xC000 – eTPU_BASE+0xCBFF[1]	3 kBytes SPRAM PSE mirror [1]
eTPU_BASE+0xCC00 – eTPU_BASE+0xFFFF	Reserved
eTPU_BASE+0x1_0000 – eTPU_BASE+1_2FFF	Shared Code Memory (SCM)[2]
eTPU_BASE+0x1_3000 – eTPU_BASE+1_FFFF	Reserved

1. Parameter sign extension access area, see Section 5.2.3, "Parameter Access."
2. SCM access is available only when bit VIS=1 on register ETPUMCR, under certain conditions (see section Section 4.3.1, "eTPU Time Base Configuration Register (ETPUTBCR)." SCM can only be written in 32 bit accesses.

Chapter 4 Programming Model

4.1 Introduction

The guideline for the description of all bits and fields throughout Chapter 4, “Programming Model,” is to provide only a brief explanation (without examples or method of use, of the features) since it will be used mainly as a reference for the reader that is studying Chapter 5, “eTPU Functional Description,” where features are explained in detail.

4.2 System Configuration Registers

4.2.1 eTPU Module Configuration Register (ETPUMCR)

This register is global to both eTPU engines, and resides in the shared BIU. ETPUMCR gathers global configuration and status in the eTPU system, including global exception. It is also used for configuring the SCM (shared code memory) operation and test.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	MGEA	MGEB	ILFA	ILFB	0	SCMSIZE						
W	GEC															
Reset	0	0	0	0	0	0	0	0	0	SCMSIZE						
Reg Addr	eTPU_BASE + 0x000															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	SCMMISF	SCMMISEN	0	0	VIS	0	0	0	0	0	GTBE
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU_BASE + 0x000															

Figure 4-1. ETPUMCR Register

Table 4-1. ETPUMCR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0	—	Reads return 0.
R/W		GEC	Global Exception Clear. This write-only bit negates global exception request and clears global exception status bits MGEA, MGEB, ILFA, ILFB and SCMMISF. A read will always return 0. Writes have the following effect: 1 Negate global exception, clear status bits ILFA, ILFB, MGEA, MGEB, and SCMMISF. 0 Keep global exception request and status bits ILFA, ILFB, MGEA, MGEB, and SCMMISF as is. GEC works the same way with either one or both engines in stop mode.
—	1 – 3	—	Writes do not affect bit values. Reads return 0.
R	4	MGEA	Microcode Global Exception Engine A. This bit indicates that a global exception was asserted by microcode executed on the respective engine. The determination of the reason why the global exception was asserted is application dependent: it can be coded in an SPRAM status parameter, for instance. This bit is cleared by writing 1 to GEC. 1 Global exception requested by microcode is pending. 0 No microcode-requested global exception pending.
R	5	MGEB	Microcode Global Exception Engine B. This bit indicates that a global exception was asserted by microcode executed on the respective engine. The determination of the reason why the global exception was asserted is application dependent: it can be coded in an SPRAM status parameter, for instance. This bit is cleared by writing 1 to GEC. 1 Global exception requested by microcode is pending. 0 No microcode-requested global exception pending.
R	6	ILFA	Illegal Instruction Flag eTPU A. The ILFA bit is set by the microengine to indicate that an illegal instruction was decoded in engine A. This bit is cleared by host writing 1 to GEC. See Section 5.9.5, “Illegal Instructions,” for more details. 1 Illegal Instruction detected by eTPU A. 0 Illegal Instruction not detected.
R	7	ILFB	Illegal Instruction Flag eTPU B. The ILFB bit is set by the microengine to indicate that an illegal instruction was decoded in engine B. This bit is cleared by host writing 1 to GEC. See Section 5.9.5, “Illegal Instructions,” for more details. 1 Illegal Instruction detected by eTPU B. 0 Illegal Instruction not detected.
—	8	—	Writes do not affect bit values. Reads return 0.
R	9 – 15	SCMSIZE[7:0]	SCM Size. This read-only field holds the number of 2 Kbyte SCM Blocks minus 1. This value is MCU-dependent.
—	16 – 20	—	Writes do not affect bit values. Reads return 0.

Table 4-1. ETPUMCR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	21	SCMMISF	SCM MISC Flag. The SCMMISF bit is set by the SCM MISC (Multiple Input Signature Calculator) logic to indicate that the calculated signature does not match the expected value, at the end of a MISC iteration. The SCMMISF bit is not affected by eTPU A or eTPU B internal reset. See Chapter 10, “Test and Development Support,” for more details. 1 MISC has read entire SCM array and the expected signature in ETPUMISCCMPR does not match the value calculated. 0 Signature mismatch not detected. This bit is automatically cleared when SCMMISEN changes from 0 to 1, or when global exception is cleared by writing 1 to GEC.
R/W	22	SCMMISEN	SCM MISC Enable. The SCMMISEN bit is used for enabling/disabling the operation of the MISC logic. SCMMISEN is readable and writable at any time. The MISC logic will only operate when this bit is set to 1. When the bit is reset the MISC address counter is set to the initial SCM address. When enabled, the MISC will continuously cycle through the SCM addresses, reading each and calculating a CRC. In order to save power, the MISC can be disabled by clearing the SCMMISEN bit. The SCMMISEN bit is not affected by eTPU A or eTPU B internal soft reset. See Section 5.10, “Test and Development Support,” for more details. 1 MISC operation enabled. (Toggling to 1 clears the SCMMISF bit) 0 MISC operation disabled. The MISC logic is reset to its initial state. SCMMISEN is cleared automatically when MISC logic detects an error; that is, when SCMMISF transitions from 0 to 1, disabling the MISC operation.
—	23 – 24	—	Writes do not affect bit values. Reads return 0.
R/W	25	VIS	SCM Visibility Bit. The VIS bit determines SCM visibility to the IP bus interface and resets the MISC state (but SCMMISEN keeps its value). 1 SCM is visible to the IP bus. The MISC state is reset. SCM is write-protected. 0 SCM is not visible to the IP bus. Accessing SCM address space issues a bus error. This bit is write protected when any of the engines is not in halt or stop states. When VIS=1, the ETPUECR STOP bits are write protected, and only 32-bit aligned SCM writes are supported. The value written to SCM is unpredictable if other transfer sizes are used.
—	26 – 30	—	Writes do not affect bit values. Reads return 0.
R/W	31	GTBE	Global Time Base Enable. GTBE enables time bases in both engines, allowing them to be started synchronously. An assertion of GTBE also starts the eMIOS time base[1]. This enables the eTPU time bases and the eMIOS time base to all start synchronously. 1 time bases in both eTPU engines and eMIOS are enabled to run. 0 time bases in both engines are disabled to run.

1. The eMIOS also has an GTBE bit. Assertion of either the eMIOS or eTPU GTBE bit starts time bases for the eMIOS and eTPU, see Section 5.6.4, “Global Time Base Enable (GTBE).”

4.2.2 eTPU Coherent Dual-Parameter Controller Register (ETPUCDCR)

ETPUCDCR configures and controls dual-parameter coherent transfers. For more info, see Section 5.4.3, "Coherent Dual-parameter Controller (CDC)."

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	STS	CTBASE					PBBASE									
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU_BASE + 0x004															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	PWIDTH	PARM0							WR	PARM1						
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU_BASE + 0x004															

Figure 4-2. ETPUCDCR Register

Table 4-2. ETPUCDCR Field Descriptions

Read/Write	Bits	Name	Description
R/W	0	STS	Start Bit. This bit is set by the host in order to start the data transfer between the parameter buffer pointed by PBBASE and the target addresses selected by the concatenation of fields CTBASE and PARM0/1. The host receives wait-states until the data transfer is complete, when this bit is reset by coherency logic, see Section 5.4.3, "Coherent Dual-parameter Controller (CDC)." Therefore, host always reads STS as 0. 1 (write) starts a coherent transfer. 0 (write) does not start a coherent transfer.
R/W	1 – 5	CTBASE[4:0]	Channel Transfer Base. This field concatenates with fields PARM0/PARM1 to determine the absolute offset (from the SPRAM base) of the parameters to be transferred: Parameter 0 address = {CTBASE, PARM0}*4 + SPRAM base Parameter 1 address = {CTBASE, PARM1}*4 + SPRAM base
R/W	6 – 15	PBBASE[9:0]	Parameter Buffer Base Address. This field points to the base address of the parameter buffer location, with granularity of 2 parameters (8 bytes). The host (byte) address of the first parameter in the buffer is PBBASE*8 + SPRAM Base Address.
R/W	16	PWIDTH	Parameter Width Selection. This bit selects the width of the parameters to be transferred between the PB and the target address. 1 Transfer 32-bit parameters. All 32 bits of the parameters are written in the destination address. 0 Transfer 24-bit parameters. The upper byte remains unchanged in the destination address.

Table 4-2. ETPUCDCR Field Descriptions

Read/Write	Bits	Name	Description
R/W	17 – 23	PARM0[6:0]	Channel Parameter number 0. This field in concatenation with CTBASE[3:0] determine the address offset (from the SPRAM base address) of the parameter which is the destination or source (defined by WR) of the coherent transfer. The SPRAM address offset of the parameter is {CTBASE, PARM0}*4.Note that PARM0 allows non-contiguous parameters to be transferred coherently[1].
R/W	24	WR	Read/Write selection. This bit selects the direction of the coherent data transfer. 1 Write operation. Data transfer is from the PB to the selected parameter RAM address. 0 Read operation. Data transfer is from the selected parameter RAM address to the PB.
R/W	25 – 31	PARM1[6:0]	Channel Parameter number 1. This field in concatenation with CTBASE[4:0] determines the address offset (from the SPRAM base) of the parameter which is the destination or source (defined by WR) of the coherent transfer. The SPRAM address offset of the parameter is {CTBASE, PARM1}*4.Note that PARM1 allows non-contiguous parameters to be transferred coherently[1].

1. The parameter pointed by {CTBASE, PARM0} is the first transferred.

4.2.3 eTPU MISC Compare Register (ETPUMISCCMPR)

ETPUMISCCMPR holds the 32-bit signature expected from the whole SCM array. This register must be written by the host with the 32-bit word to be compared against the calculated signature at the end of the MISC cycle. This register is global to both eTPU engines. For more detail see Section 5.10.3.1, "SCM Test for MISC (Multiple Input Signature Calculator)."

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	ETPUMISCCMP[0:15]															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU_BASE + 0x00C															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	ETPUMISCCMP[16:31]															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU_BASE + 0x00C															

Figure 4-3. ETPUMISCCMPR Register

Table 4-3. ETPUMISCCMPR Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	0 – 31	ETPUMISCCMP[31:0]	Expected Multiple Input Signature Register value. See Section 5.10.3.1, “SCM Test for MISC (Multiple Input Signature Calculator).”

4.2.4 eTPU Engine Configuration Register (ETPUECR)

Each engine has its own ETPUECR register. ETPUECR holds configuration and status fields that are programmed independently in each engine.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	FEND	STOP	0	STF	0	0	0	0	HLTF	0	0	0	0		FPSCK	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x014 / eTPU B: eTPU_BASE + 0x018															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	CDFC		0	0	0	0	0	0	0	0	0			ETB		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x014 / eTPU B: eTPU_BASE + 0x018															

Figure 4-4. ETPUECR Register

Table 4-4. ETPUECR Field Descriptions

Read/ Write	Bits	Name	Description
R/W	0	FEND	Force End. FEND assertion terminates any current running thread as if an END instruction have been executed, see Section 5.9.4.1, “Ending Current Thread (END).” 1 Puts engine in reset. 0 Normal operation. This bit is self-negating during the access, i.e., the host receives wait-states while the reset occurs, and FEND always reads as 0. FEND assertion is ignored when the microengine is in TST, Halt, or Idle.
R/W	1	STOP	Low Power Stop Bit. When STOP is set, the engine shuts down its internal clocks. TCR1 and TCR2 cease to increment, and input sampling stops. The engine asserts the stop flag (STF) bit to indicate that it has stopped However, the BIU continues to run, and the host can access all registers except for the channel registers[1] (see list of channel registers on Section 4.6, “Channel Configuration and Control Registers.” After STOP is set, even before STF asserts, data read from the channel registers is not meaningful, a Bus Error is issued, and writes are ineffective. When the STOP bit is asserted while the microcode is executing, the eTPU will stop when the thread is complete. 1 Commands engine to stop its clocks. 0 eTPU engine runs. Stop completes on the next system clock after the stop condition is valid. The STOP bit is write-protected when VIS=1.
—	2	—	Writes do not affect bit values. Reads return 0.
R	3	STF	Stop Flag Bit. Each engine asserts its stop flag (STF) to indicate that it has stopped. Only then the host can assume that the engine has actually stopped. The eTPU system is fully stopped when the STF bits of both eTPU engines are asserted. In case of STAC bus stop, the eTPU system responds with stop acknowledge only after both eTPU A and eTPU B have been stopped. The engine only stops when any ongoing thread is complete also in this case. 1 The engine has stopped (after the local STOP bit has been asserted, or after the STAC bus stop line has been asserted). 0 The engine is operating. Summarizing engine stop conditions, which STF reflects: STF_A := (after stop completed) STOP_A STF_B := (after stop completed) STOP_B STF_A and STF_B mean STF bit from engine A and STF bit from engine B respectively.
—	4 – 7	—	Writes do not affect bit values. Reads return 0.
R	8	HLTF	Halt Mode Flag. If eTPU engine entered halt state, this flag is asserted. The flag remains asserted while the microengine is in halt state, even during a single-step or forced instruction execution. See Section 10.2, “Development Support Features,” for further details about entering halt mode. 1 eTPU engine is halted 0 eTPU engine is not halted.
—	9 – 12	—	Writes do not affect bit values. Reads return 0.
R/W	13 – 15	FPSCK[2:0]	Filter Prescaler Clock Control. FPSCK controls the prescaling of the clocks used in digital filters for the channel input signals and TCRCLK input, as shown in Table 4-5. Filtering can be controlled independently by the engine, but all input digital filters in the same engine have same clock prescaling. For more details see Section 5.5.6.4, “Filter Clock Prescaler.”

Table 4-4. ETPUECR Field Descriptions

Read/ Write	Bits	Name	Description
R/W	16 – 17	CDFC[1:0]	Channel Digital Filter Control. These bits select a digital filtering mode for the channels when configured as inputs for improved noise immunity (refer toTable 4-6). The eTPU has three digital filtering modes for the channels which provide programmable trade-off between signal latency and noise immunity, see Section 5.5.6, “Enhanced Digital Filter (EDF).” Changing CDFC during eTPU normal input channel operation is not recommended since it changes the behavior of the transition detection logic while executing its operation.
—	18 – 26	—	Writes do not affect bit values. Reads return 0.
R/W	27 – 31	ETB[4:0]	Entry Table Base. The field determines the location of the microcode entry table for the eTPU functions in SCM, see Section 7.2, “Entry Points.” Table 4-7 shows the entry table base address options.

1. The time base registers can still be read in Stop mode, but writes are unpredictable and a Bus Error is issued. Global channel registers and SPRAM can be accessed normally.

Table 4-5. Filter Prescaler Clock Control

Filter Control	Sample on System Clock Divided by:
000	2
001	4
010	8
011	16
100	32
101	64
110	128
111	256

Table 4-6. Channel Digital Filter Control

CDFC	Selected Digital Filter
00	TPU2/3 Two Sample Mode: Using the filter clock which is the system clock divided by (2, 4, 8,..., 256) as a sampling clock (selected by FPSCK field in ETPUECR), comparing two consecutive samples which agree with each other sets the input signal state. This is the default reset state.
01	RESERVED

Table 4-6. Channel Digital Filter Control

CDFC	Selected Digital Filter
10	eTPU Three Sample Mode: Similar to the TPU2/3 two sample mode, but comparing three consecutive samples which agree with each other sets the input signal state.
11	eTPU Continuous Mode: Signal needs to be stable for the whole filter clock period. This mode compares all the values at the rate of system clock divided by two, between two consecutive filter clock pulses. Signal needs to be continuously stable for the entire period. If all the values agree with each other, input signal state is updated.

Table 4-7. Entry Table Base Address Options

ETB	Entry Table Base Address for CPU Host Address (byte format)	Entry Table Base Address for Microcode Address (word format)
00000	0x000	0x000
00001	0x800	0x200
00010	0x1000	0x400
. . . .	. . . .	. . . .
11110	0xF000	0x3C00
11111	0xF800	0x3E00

4.3 Time Base Registers

Time base registers allows the configuration and visibility of internally-generated time bases TCR1 and TCR2. There is one of each of these registers for each eTPU engine.

4.3.1 eTPU Time Base Configuration Register (ETPUTBCR)

This register configures several time base options.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	TCR2CTL			TCRCF		0	AM	0	0	0	TCR2P					
W						▨		▨	▨	▨						
Reset	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x020 / eTPU B: eTPU_BASE + 0x040															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	TCR1CTL		0	0	0	0	0	0	TCR1P							
W			▨	▨	▨	▨	▨	▨								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x020 / eTPU B: eTPU_BASE + 0x040															

Figure 4-5. ETPUTBCR Register

Table 4-8. ETPUTBCR Field Descriptions

Read/ Write	Bits	Name	Description
R/W	0 – 2	TCR2CTL[2:0]	TCR2 Clock/Gate Control. These bits are part of the TCR2 clocking system, see Section 5.6, “Time Bases.” They determine the clock source for TCR2. TcR2 can count on any detected edge of the TCRCLK signal or use it for gating system clock divided by 8. After reset, TCRCLK signal rising edge is selected. TCR2 can also be clocked by the system clock divided by 8.
R/W	3 – 4	TCRCF[1:0]	TCRCLK Signal Filter Control. This field controls the TCRCLK digital filter, see Section 5.6.5, “TCRCLK Digital Filter,” determining whether the TCRCLK signal input (after a synchronizer) is filtered with the same filter clock as the channel input signals, see Section 5.5.6, “Enhanced Digital Filter (EDF),” or uses the system clock divided by 2, and also whether the TCRCLK digital filter works in integrator mode or two sample mode, see Table 4-10. For information on integration mode see Section 5.6.5, “TCRCLK Digital Filter.” For information on two sample mode see Section 5.5.6.1, “Two-Sample Mode.”
—	5	—	Writes do not affect bit values. Reads return 0.
R/W	6	AM	Angle Mode Selection. When the AM bit is set and neither TCR1 nor TCR2 are STAC interface clients, the EAC (eTPU Angle Clock) hardware provides angle information to the channels using the TCR2 bus. When the AM is reset (non-angle mode), EAC operation is disabled, and its internal registers can be used as general purpose registers. For more information, see Section 5.7, “eTPU Angle Counter (EAC).” 1 TCR2 works in angle mode. If TCR2 is not a STAC client, see Section 5.6.3, “Shared Time and Angel Count (STAC) Bus Interface,” the EAC works and stores Tooth Counter and Angle Tick Counter data in TCR2. 0 EAC operation is disabled. If TCR1 or TCR2 is a STAC bus client, EAC operation is forbidden. Therefore, if AM is set, the angle logic does not work properly.
—	7 – 9	—	Writes do not affect bit values. Reads return 0.

Table 4-8. ETPUTBCR Field Descriptions

Read/ Write	Bits	Name	Description
R/W	10 – 15	TCR2P[5:0]	Timer Count Register 2 Prescaler Control. These bits are part of the TCR2 clocking system, see Section 5.6, “Time Bases.” TCR2 is clocked from the output of a prescaler. The prescaler divides its input by (TCR2P+1) allowing frequency divisions from 1 to 64. The prescaler input is the system clock divided by 8 (in gated or non-gated clock mode) or TCRCLK filtered input.
R/W	16 – 17	TCR1CTL[1:0]	TCR1 Clock/Gate Control. TCR1CTL is part of the TCR1 clocking system, see Section 5.6, “Time Bases” It determines the clock source for TCR1. TCR1 can count on detected rising edge of the TCRFCLK signal or the system clock divided by 2, see Table 4-11. After reset TCRCLK signal is selected.
—	18 – 23	—	Writes do not affect bit values. Reads return 0.
R/W	24 – 31	TCR1P[7:0]	Timer Count Register 1 Prescaler Control. TCR1 is clocked from the output of a prescaler. The input to the prescaler is the internal eTPU system clock divided by 2 or the output of TCRCLK filter. The prescaler divides this input by (TCR1P+1) allowing frequency divisions from 1 up to 256.

Table 4-9. TCR2 Clock Source

TCR2CTL	TCR2 Clock	Angle Tooth Detection
000	Gated DIV8 clock (system clock / 8). When the external TCRCLK signal is low, the DIV8 clock is blocked, preventing it from incrementing TCR2. When the external TCRCLK signal is high, TCR2 is incremented at the frequency of the system clock divided by 8.	N/A[1]
001	Rise transition on TCRCLK signal increments TCR2.	Rising Edge
010	Fall transition on TCRCLK signal increments TCR2.	Falling Edge
011	Rise or fall transition on TCRCLK signal increments TCR2.	Both
100	DIV8 clock (system clock / 8)	N/A[1]
101	Reserved	N/A[1]
110		
111	TCR2CTL shuts down TCR2 clocking, except on Angle Mode. TCR2 can also change as STAC client.	

These selections must not be used in Angle Mode.

Table 4-10. TCRCLK Filter Clock/Mode

TCRCF	Filter Input	Filter Mode
00	system clock divided by 2	two sample
01	filter clock of the channels	two sample
10	system clock divided by 2	integration
11	filter clock of the channels	integration

Table 4-11. TCR1 Clock Source

TCR1CTL	TCR1 Clock
00	selects TCRCLK as clock source for the TCR1 prescaler
01	reserved
10	selects system clock divided by 2 as clock source for the TCR1 prescaler
11	TCR1CTL shuts down TCR1 clock. TCR1 can still change if STAC client.

4.3.2 eTPU Time Base 1 (TCR1) Visibility Register (ETPUTB1R)

This register provides visibility of the TCR1 time base for CPU host read access, see Section 5.6, “Time Bases.” This register is read-only. The value of the TCR1 time base shown can be driven by the TCR1 counter or imported, depending on the configuration set in ETPUREDCR.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	TCR1[23:16]							
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr eTPU A: eTPU_BASE + 0x024 / eTPU B: eTPU_BASE + 0x044

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	TCR1[15:0]															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr eTPU A: eTPU_BASE + 0x024 / eTPU B: eTPU_BASE + 0x044

Figure 4-6. ETPUTB1R Register

Table 4-12. ETPUTB1R Bit Field Descriptions

Read/Write	Bits	Name	Description
—	0 – 7	—	Writes do not affect bit values. Reads return 0.
R	8 – 31	TCR1[23:0]	TCR1 value. TCR1 value used on matches and captures. See Section 5.6, “Time Bases.”

4.3.3 eTPU Time Base 2 (TCR2) Visibility Register (ETPUTB2R)

This register provides visibility of the TCR2 time base for CPU host read access, see Section 5.6, “Time Bases.”. This register is read-only. The value of the TCR2 time base shown can be driven by the TCR2 counter, the angle mode logic, or imported from the

STAC interface, depending on angle mode (an engine cannot import when in angle mode) and STAC interface configurations set in registers ETPUTBCR and ETPUREDCR.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	TCR2[23:16]							
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x028 / eTPU B: eTPU_BASE + 0x048															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	TCR2[15:0]															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x028 / eTPU B: eTPU_BASE + 0x048															

Figure 4-7. ETPUTB2R Register

Table 4-13. ETPUTB2R Bit Field Descriptions

Read/Write	Bits	Name	Description
—	0 – 7	—	Writes do not affect bit values. Reads return 0.
R	8 – 31	TCR2[23:0]	TCR2 value. TCR2 value used on matches and captures. See Section 5.6, “Time Bases.”

4.3.4 STAC Bus Configuration Register (ETPUREDCR)

This register configures the eTPU STAC bus interface module and operation, see Section 5.6.3, “Shared Time and Angel Count (STAC) Bus Interface.”.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	REN1	RSC1	0	0	SERVER_ID1				0	0	0	0	SRV1			
W																
RESET:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADDR	eTPU A: eTPU_BASE + 0x02C / eTPU B: eTPU_BASE + 0x04C															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	REN2	RSC2	0	0	SERVER_ID2				0	0	0	0	SRV2			
W																
RESET:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ADDR	eTPU A: eTPU_BASE + 0x02C / eTPU B: eTPU_BASE + 0x04C															

Figure 4-8. ETPUREDCR Register

Table 4-14. ETPUREDCR Field Descriptions

Read/ Write	Bits	Name	Description
R/W	0	REN1	TCR1 Resource[1] Client/Server Operation Enable Bit. This bit enables or disables client/server operation to eTPU STAC interface. REN1 enables TCR1. 1 Server/Client Operation for resource 1 is enabled. 0 Server/Client Operation for resource 1 is disabled.
R/W	1	RSC1	TCR1 Resource Server/Client Assignment Bit. This bit selects the eTPU data resource assignment to be used as a Server or Client. RSC1 selects the functionality of TCR1. For Server mode, external plugging determines the unique server address assigned to each TCR. For a Client mode, the SRV1 field determines the Server address to which the Client listens. 1 Resource Server operation. 0 Resource Client operation.
—	2 - 3	—	Writes do not affect bit values. Reads return 0.
R	4 - 7	SERVER_ID1	SERVER_ID1 returns the STAC bus "addresses" for TCR1 when configured as a server. The values are: 0 and 2 for engine 1 1 and 3 for engine 2
—	8 - 11	—	Writes do not affect bit values. Reads return 0.
R/W	12 – 15	SRV1[3:0]	TCR1 Resource Server. These bits select the address of the specific IP Server to which the local TCR1 listens when configured as a IP Client. SRV1 selects the IP Server of TCR1.
R/W	16	REN2	TCR2 Resource[1] Client/Server Operation Enable Bit. This bit enables or disables Client/Server operation to eTPU IP resources. REN2 enables TCR2 IP bus operations. 1 Server/Client Operation for resource 2 is enabled. 0 Server/Client Operation for resource 2 is disabled.
R/W	17	RSC2	TCR2[2] Resource Server/Client Assignment Bit. This bit selects the eTPU data resource assignment to be used as a Server or Client. RSC2 selects the functionality of TCR2. For Server mode, external plugging determines the unique server address assigned to each TCR. For a Client mode, the SRV2 field determines the Server address to which the Client listens. 1 Resource Server operation. 0 Resource Client operation.
—	18 - 19	—	Writes do not affect bit values. Reads return 0.
R	20 - 23	SERVER_ID2	SERVER_ID2 returns the STAC bus "addresses" for TCR2 when configured as a server. The values are: 0 and 2 for engine 1 1 and 3 for engine 2
—	24 - 27	—	Writes do not affect bit values. Reads return 0.
R/W	28 – 31	SRV2[3:0]	TCR2 Resource Server. These bits select the address of the specific IP Server to which the local TCR2 listens when configured as a IP Client. SRV2 selects the IP Server of TCR2.

1. Resource identifies any parameter that changes along the time and can be exported / imported from other device. In eTPU context, a resource can be TCR1, or TCR2 (either time or angle values).

2. When TCR2 is configured as a IP bus client (REN2=1, RSC2=0) the eTPU angle clock hardware is disabled. In this case the AM (angle mode) bit in ETPUTBCR has no effect.

4.4 Channel Registers Layout

The channel register structure map is shown in Table 4-15. Every eTPU channel has 3 registers that fit within the address range of each structure; the 3 registers are illustrated in Table 4-16.

Table 4-15. eTPU Channel Register Map

Address	Registers Structure
eTPU_BASE+0x400	eTPU A Channel 0 Register Structure
eTPU_BASE+0x410	eTPU A Channel 1 Register Structure
eTPU_BASE+0x420	eTPU A Channel 2 Register Structure
eTPU_BASE+0x430 – eTPU_BASE+0x5D0	. . .
eTPU_BASE+0x5E0	eTPU A Channel 30 Register Structure
eTPU_BASE+0x5F0	eTPU A Channel 31 Register Structure
eTPU_BASE+0x600 – eTPU_BASE+0x7FF	Reserved
eTPU_BASE+0x800	eTPU B Channel 0 Register Structure
eTPU_BASE+0x810	eTPU B Channel 1 Register Structure
eTPU_BASE+0x820	eTPU B Channel 2 Register Structure
eTPU_BASE+0x430 – eTPU_BASE+0x5D0	. . .
eTPU_BASE+0x9E0	eTPU B Channel 30 Register Structure
eTPU_BASE+0x9F0	eTPU B Channel 31 Register Structure
eTPU_BASE+0xA00 – eTPU_BASE+0xBFF	Reserved

Table 4-16. eTPU Channel Registers Structure

Offset	Register Name
0x00	eTPU Channel Configuration Register (eTPUCxCCR)
0x04	eTPU Channel Status/Control Register (eTPUCSCR)
0x08	eTPU Channel Host Service Request Register (eTPUCHSRR)
0x0c	Reserved

4.5 Global Channel Registers

The registers in this section group, by type, the interrupt status and enable bits from all the channels. This organization eases management of all channels or groups of channels by a single interrupt handler routine. These bits are mirrored in the individual channel registers, grouped by channel.

4.5.1 eTPU Channel Interrupt Status Register (ETPUCISR)

Host interrupt status, see Section 5.2.2, “Interrupts and Data Transfer Requests, from all channels are grouped in ETPUCISR. Their bits are mirrored from the channel status/control registers, see Section 4.6, “Channel Configuration and Control Registers,” and the host CPU must write 1 to clear a status bit.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	CIS31	CIS30	CIS29	CIS28	CIS27	CIS26	CIS25	CIS24	CIS23	CIS22	CIS21	CIS20	CIS19	CIS18	CIS17	CIS16
W	CIC31	CIC30	CIC29	CIC28	CIC27	CIC26	CIC25	CIC24	CIC23	CIC22	CIC21	CIC20	CIC19	CIC18	CIC17	CIC16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x200 / eTPU B: eTPU_BASE + 0x204															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	CIS15	CIS14	CIS13	CIS12	CIS11	CIS10	CIS9	CIS8	CIS7	CIS6	CIS5	CIS4	CIS3	CIS2	CIS1	CIS0
W	CIC15	CIC14	CIC13	CIC12	CIC11	CIC10	CIC9	CIC8	CIC7	CIC6	CIC5	CIC4	CIC3	CIC2	CIC1	CIC0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x200 / eTPU B: eTPU_BASE + 0x204															

Figure 4-9. ETPUCISR Register

Table 4-17. ETPUCISR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0 – 31	CISx	Channel x Interrupt Status. 1 indicates that channel x has a pending interrupt to the host CPU. 0 indicates that channel x has no pending interrupt to the host CPU.
W	0 – 31	CICx	Channel x Interrupt Clear 1 clear interrupt status bit. 0 keep interrupt status bit unaltered. For details about interrupts see Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”

4.5.2 eTPU Channel Data Transfer Request Status Register (ETPUCDTRSR)

Data transfer request status, see Section 5.2.2, “Interrupts and Data Transfer Requests,” from all channels are grouped in ETPUCDTRSR. Their bits are mirrored from the channel status/control registers, see Section 4.6.2, “eTPU Channel x Status Control Register (ETPUCxSCR).”

NOTE

eTPU A channels [0:2,12:15,28:29] and eTPU B channels [0:3,12:15,28:31] are connected to the DMA. The data transfer request lines that are not connected to the DMA controller are left disconnected and don’t generate interrupt requests, even if their request status bits are asserted in registers ETPUCDTRSR and ETPUCxSCR. Channels that are not connected may still have their status bits (DTRS*x*) cleared by writing to the appropriate field (DTRC*x*)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	DTRS 31	DTRS 30	DTRS 29	DTRS 28	DTRS 27	DTRS 26	DTRS 25	DTRS 24	DTRS 23	DTRS 22	DTRS 21	DTRS 20	DTRS 19	DTRS 18	DTRS 17	DTRS 16
W	DTRC 31	DTRC 30	DTRC 29	DTRC 28	DTRC 27	DTRC 26	DTRC 25	DTRC 24	DTRC 23	DTRC 22	DTRC 21	DTRC 20	DTRC 19	DTRC 18	DTRC 17	DTRC 16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x210 / eTPU B: eTPU_BASE + 0x214															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	DTRS 15	DTRS 14	DTRS 13	DTRS 12	DTRS 11	DTRS 10	DTRS 9	DTRS 8	DTRS 7	DTRS 6	DTRS 5	DTRS 4	DTRS 3	DTRS 2	DTRS 1	DTRS 0
W	DTRC 15	DTRC 14	DTRC 13	DTRC 12	DTRC 11	DTRC 10	DTRC 9	DTRC 8	DTRC 7	DTRC 6	DTRC 5	DTRC 4	DTRC 3	DTRC 2	DTRC 1	DTRC 0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x210 / eTPU B: eTPU_BASE + 0x214															

Figure 4-10. ETPUCDTRSR Register

Table 4-18. ETPUCDTRSR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0 – 31	DTRSx	Channel x Data Transfer Request Status. 1 indicates that channel x has a pending data transfer request. 0 indicates that channel x has no pending data transfer request.
W	0 – 31	DTRCx	Channel x Data Transfer Request Clear. 1 clear status bit. 0 keep status bit unaltered For details about interrupts see Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”

4.5.3 eTPU Channel Interrupt Overflow Status Register (ETPUCIOSR)

Interrupt Overflow status, see Section 5.2.2, “Interrupts and Data Transfer Requests,” from all channels are grouped in ETPUCIOSR. Their bits are mirrored from the channel status/control registers, see Section 4.6.2, “eTPU Channel x Status Control Register (ETPUCxSCR),” and the host must write 1 to clear a status bit.

NOTE

An interrupt overflow occurs when an interrupt is issued for a channel when the previous interrupt status bit for the same channel has not been cleared.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	CIOS 31	CIOS 30	CIOS 29	CIOS 28	CIOS 27	CIOS 26	CIOS 25	CIOS 24	CIOS 23	CIOS 22	CIOS 21	CIOS 20	CIOS 19	CIOS 18	CIOS 17	CIOS 16
W	CIOC 31	CIOC 30	CIOC 29	CIOC 28	CIOC 27	CIOC 26	CIOC 25	CIOC 24	CIOC 23	CIOC 22	CIOC 21	CIOC 20	CIOC 19	CIOC 18	CIOC 17	CIOC 16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x220 / eTPU B: eTPU_BASE + 0x224															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	CIOS 15	CIOS 14	CIOS 13	CIOS 12	CIOS 11	CIOS 10	CIOS 9	CIOS 8	CIOS 7	CIOS 6	CIOS 5	CIOS 4	CIOS 3	CIOS 2	CIOS 1	CIOS 0
W	CIOC 15	CIOC 14	CIOC 13	CIOC 12	CIOC 11	CIOC 10	CIOC 9	CIOC 8	CIOC 7	CIOC 6	CIOC 5	CIOC 4	CIOC 3	CIOC 2	CIOC 1	CIOC 0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x220 / eTPU B: eTPU_BASE + 0x224															

Figure 4-11. ETPUCIOSR Register

Table 4-19. ETPUCIOSR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0 – 31	CIOSx	Channel x Interrupt Overflow Status. 1 indicates that interrupt overflow occurred in the channel. 0 indicates that no interrupt overflow occurred in the channel.
W	0 – 31	CIOCx	Channel x Interrupt Overflow Clear. 1 clear status bit. 0 keep status bit unaltered. For details about interrupt overflow, see Section 5.2.2.2, “Interrupt and Data Transfer Request Overflow.”

4.5.4 eTPU Channel Data Transfer Request Overflow Status Register (ETPUCDTROSR)

Data transfer request overflow status, see Section 5.2.2, “Interrupts and Data Transfer Requests,” from all channels are grouped in ETPUCDTROSR. Their bits are mirrored from the channel status/control registers, see Section 4.6.2, “eTPU Channel x Status Control Register (ETPUCxSCR),” and the host must write 1 to clear a status bit.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	DTR OS 31	DTR OS 30	DTR OS 29	DTR OS 28	DTR OS 27	DTR OS 26	DTR OS 25	DTR OS 24	DTR OS 23	DTR OS 22	DTR OS 21	DTR OS 20	DTR OS 19	DTR OS 18	DTR OS 17	DTR OS 16
W	DTR OC 31	DTR OC 30	DTR OC 29	DTR OC 28	DTR OC 27	DTR OC 26	DTR OC 25	DTR OC 24	DTR OC 23	DTR OC 22	DTR OC 21	DTR OC 20	DTR OC 19	DTR OC 18	DTR OC 17	DTR OC 16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr eTPU A: eTPU_BASE + 0x230 / eTPU B: eTPU_BASE + 0x234

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	DTR OS 15	DTR OS 14	DTR OS 13	DTR OS 12	DTR OS 11	DTR OS 10	DTR OS 9	DTR OS 8	DTR OS 7	DTR OS 6	DTR OS 5	DTR OS 4	DTR OS 3	DTR OS 2	DTR OS 1	DTR OS 0
W	DTR OC 15	DTR OC 14	DTR OC 13	DTR OC 12	DTR OC 11	DTR OC 10	DTR OC 9	DTR OC 8	DTR OC 7	DTR OC 6	DTR OC 5	DTR OC 4	DTR OC 3	DTR OC 2	DTR OC 1	DTR OC 0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr eTPU A: eTPU_BASE + 0x230 / eTPU B: eTPU_BASE + 0x234

Figure 4-12. ETPUCDTROSR Register

Table 4-20. ETPUCDTROSR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0 – 31	DTROSx	Channel x Data Transfer Request Overflow Status. 1 indicates that data transfer request overflow occurred in the channel. 0 indicates that no data transfer request overflow occurred in the channel.
W	0 – 31	DTROCx	Channel x Data Transfer Request Overflow Clear. 1 clear status bit. 0 keep status bit unaltered. For details about data transfer request overflow, see Section 5.2.2.2, “Interrupt and Data Transfer Request Overflow.”

4.5.5 eTPU Channel Interrupt Enable Register (ETPUCIER)

Host interrupt enable, see Section 5.2.2, “Interrupts and Data Transfer Requests,” from all channels are grouped in ETPUCIER. Their bits are mirrored from the channel configuration registers, see Section 4.6.1, “eTPU Channel x Configuration Register (ETPUCxCR)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	CIE 31	CIE 30	CIE 29	CIE 28	CIE 27	CIE 26	CIE 25	CIE 24	CIE 23	CIE 22	CIE 21	CIE 20	CIE 19	CIE 18	CIE 17	CIE 16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x240 / eTPU B: eTPU_BASE + 0x244															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	CIE 15	CIE 14	CIE 13	CIE 12	CIE 11	CIE 10	CIE 9	CIE 8	CIE 7	CIE 6	CIE 5	CIE 4	CIE 3	CIE 2	CIE 1	CIE 0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x240 / eTPU B: eTPU_BASE + 0x244															

Figure 4-13. ETPUCIER Register

Table 4-21. ETPUCIER Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	0 – 31	CIEx	Channel x Interrupt Enable. 1 interrupt enabled for channel x 0 interrupt disabled for channel x. For details about interrupts see Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”

4.5.6 eTPU Channel Data Transfer Request Enable Register (ETPUCDTRER)

Data transfer request enable, see Section 5.2.2, “Interrupts and Data Transfer Requests,” from all channels are grouped in ETPUCDTRER. These bits are mirrored from the channel configuration registers, see Section 4.6.1, “eTPU Channel x Configuration Register (ETPUCxCR)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	DTRE 31	DTRE 30	DTRE 29	DTRE 28	DTRE 27	DTRE 26	DTRE 25	DTRE 24	DTRE 23	DTRE 22	DTRE 21	DTRE 20	DTRE 19	DTRE 18	DTRE 17	DTRE 16
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr eTPU A: eTPU_BASE + 0x250 / eTPU B: eTPU_BASE + 0x254

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	DTRE 15	DTRE 14	DTRE 13	DTRE 12	DTRE 11	DTRE 10	DTRE 9	DTRE 8	DTRE 7	DTRE 6	DTRE 5	DTRE 4	DTRE 3	DTRE 2	DTRE 1	DTRE 0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr eTPU A: eTPU_BASE + 0x250 / eTPU B: eTPU_BASE + 0x254

Figure 4-14. ETPUCDTRER Register

Table 4-22. ETPUCDTRER Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	0 – 31	DTREx	Channel x Data Transfer Request Enable. 1 Data Transfer request enabled for channel x. 0 Data Transfer request disabled for channel x. For details about interrupts see Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”

4.5.7 eTPU Channel Pending Service Status Register (ETPUCPSSR)

ETPUCPSSR is a read-only register that holds the status of the pending channel service requests, see Section Chapter 7, “Functions and Threads.”

NOTE

More than one source may be requesting service when a channel’s service request bit is set.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	SR31	SR30	SR29	SR28	SR27	SR26	SR25	SR24	SR23	SR22	SR21	SR20	SR19	SR18	SR17	SR16
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x280 / eTPU B: eTPU_BASE + 0x284															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	SR15	SR14	SR13	SR12	SR11	SR10	SR9	SR8	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x280 / eTPU B: eTPU_BASE + 0x284															

Figure 4-15. ETPUCPSSR Register

Table 4-23. ETPUCPSSR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0 – 31	SRx	Pending Service Request x. Indicates a pending service request for channel x. 1 pending service request for channel x 0 no service request pending for channel x Pending SR status is negated at the time slot transition to the respective service thread.

NOTE

The pending service status bit for a channel is set when a service request is pending, even if the Channel is disabled (CPRx = 0).

4.5.8 eTPU Channel Service Status Register (ETPUCSSR)

ETPUCSSR holds the current channel service status on whether it is being serviced or not, see Section Chapter 7, “Functions and Threads.” Only one bit may be asserted in this register at a given time. When no channel is being serviced the register read value is 0x00000000. ETPUCSSR is a read-only register. The register can be read during normal eTPU operation for monitoring the scheduler activity.

NOTE

The ETPUCSSR is not an absolute indication of channel status. If more than one source is requesting service, the asserted status bit only indicates that one of the requests has been granted.

NOTE

Channel service status does not always reflect decoding of the CHAN register, since the later can be changed by the service thread microcode.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	SS31	SS30	SS29	SS28	SS27	SS26	SS25	SS24	SS23	SS22	SS21	SS20	SS19	SS18	SS17	SS16
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x290 / eTPU B: eTPU_BASE + 0x294															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	SS15	SS14	SS13	SS12	SS11	SS10	SS9	SS8	SS7	SS6	SS5	SS4	SS3	SS2	SS1	SS0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	eTPU A: eTPU_BASE + 0x290 / eTPU B: eTPU_BASE + 0x294															

Figure 4-16. ETPUCSSR Register

Table 4-24. ETPUCSSR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0 – 31	SSx	Service Status x. Indicates that channel x is currently being serviced. It is updated at the 1st microcycle of a time slot transition. See Section 7.3, “Time Slot Transition,” for more information on time slot transitions. 1 channel x is currently being serviced 0 channel x is not currently being serviced

4.6 Channel Configuration and Control Registers

Each channel, for both eTPU engines, has a group of 3 registers used to control, configure and check status of that channel as shown in Table 4-25. This organization eases individual channel management.

Table 4-25. Channel Registers Structure

Channel Offset	Register Name
0x00	eTPU Channel Configuration Register (ETPUCxCR)
0x04	eTPU Channel Status/Control Register[1] (ETPUCxSCR)
0x08	eTPU Channel Host Service Request Register (ETPUCxHSRR)
0x0C	RESERVED

1. eTPU A channels [0:2,12:15,28:29] and eTPU B channels [0:3,12:15,28:31] are connected to the DMA. The data transfer request lines that are not connected to the DMA controller are left disconnected and don't generate interrupt requests, even if their request status bits assert in registers ETPUCDTRSR and ETPUCxSCR

One contiguous area is used to map all channel registers of each eTPU engine as shown inTable 4-26.

Table 4-26. Channel Registers Map

Offset	Registers Structure
0x400	eTPU A Channel 0 Registers Structure
0x410	eTPU A Channel 1 Registers Structure
0x420	eTPU A Channel 2 Registers Structure
0x430 0x5D0	. . .
0x5E0	eTPU A Channel 30 Registers Structure
0x5F0	eTPU A Channel 31 Registers Structure
0x600	RESERVED
0x800	eTPU B Channel 0 Registers Structure
0x810	eTPU B Channel 1 Registers Structure
0x820 0x9D0	. . .
0x9E0	eTPU B Channel 30 Registers Structure
0x9F0	eTPU B Channel 31 Registers Structure
0xA00	RESERVED

There are 64 structures defined, one for each available channel in the eTPU System (32 for each engine). The base address for the structure presented can be calculated by using the following equation:

Channel_Register_Structure_Base_Address =

ETPU_Engine_Channel_Base + (channel_number * 0x10)

where:

ETPU_Engine_Channel_Base = ETPU_Base + (0x400 for engine A or 0x800 for engine B).

4.6.1 eTPU Channel x Configuration Register (ETPUCxCR)

ETPUCxCR gathers configurations set individually per channel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	CIE	DTRE	CPR		0	0	0	ETCS	0	0	0	CFS				
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	Channel_Register_Base + 0x0															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	ODIS	OPOL	0	0	0	CPBA										
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	Channel_Register_Base + 0x0															

Figure 4-17. ETPUCxCR Register

Table 4-27. ETPUCxCR Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	0	CIE	Channel Interrupt Enable. This bit is mirrored from ETPUCIER, see Section 4.5.5, “eTPU Channel Interrupt Enable Register (ETPUCIER).” 1 Enable interrupt for this channel. 0 Disable interrupt for this channel. See Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”
R/W	1	DTRE	Channel Data Transfer Request Enable. This bit is mirrored from ETPUCDTRER, see Section 4.5.6, “eTPU Channel Data Transfer Request Enable Register (ETPUCDTRER).” 1 Enable data transfer request for this channel. 0 Disable data transfer request for this channel. See Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”
R/W	2 – 3	CPR[1:0]	Channel Priority. This field defines the priority level, see Table 4-28, for the channel. This level is used by the hardware scheduler, see Section 5.3, “Scheduler.”
—	4 – 6	—	Writes do not affect bit values. Reads return 0.
R/W	7	ETCS	Entry Table Condition Select. This bit determines the channel condition encoding scheme that selects, according to channel conditions, the entry point to be taken in an entry table. ETCS value has to be compatible with the function chosen for the channel, selected in field CFS. Two condition encoding schemes are available. For details about entry table and condition encoding schemes, refer to Section 7.2, “Entry Points.” 1 select alternate entry table condition encoding scheme. 0 select standard entry table condition encoding scheme.
—	8 – 10	—	Writes do not affect bit values. Reads return 0.
R/W	11 – 15	CFS[4:0]	Channel Function Select. This field defines the function to be performed by the channel, see Section Chapter 7, “Functions and Threads.” The function assigned to the channel has to be compatible with the channel condition encoding scheme, selected by field ETCS.

Table 4-27. ETPUCxCR Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	16	ODIS	Output Disable. This bit enables the channel to have its output forced to the value opposite to OPOL when the output disable input signal corresponding to the channel group that it belongs is active. See Section 2.2.3, “Channel Output Disable Signals.” 1 turns on the output disable feature for the channel 0 turns off the output disable feature for the channel.
R/W	17	OPOL	Output Polarity. Determines the output signal polarity. The activation of the output disable signal forces, when enabled by the ODIS bit, the channel output signal to the opposite of this polarity. 1 output active high (output disable drives output to low) 0 output active low (output disable drives output to high)
—	18 – 20	—	Writes do not affect bit values. Reads return 0.
R/W	21 – 31	CPBA[10:0]	Channel x Parameter Base Address. The value of this field multiplied by 8 specifies the SPRAM parameter base host (byte) address for channel x (2-parameter granularity); for more information on SPRAM addresses, see Section 5.2.4, “SPRAM Organization.” As seen by the host, the channel parameter base (byte) address is: • without parameter sign extension: eTPU_Base + 0x8000 + CPBA*8 • with parameter sign extension: eTPU_Base + 0xC000 + CPBA*8

Table 4-28. Priority level Bits

CPR	Priority
00	Disabled
01	Low
10	Middle
11	High

4.6.2 eTPU Channel x Status Control Register (ETPUCxSCR)

ETPUCxSCR gathers the interrupt status bits of the channel, and also the function mode definition (read-write). Bits CIS, CIOS and DTRS for each channel can be also accessed from ETPUCISR, ETPUCIOSR and ETPUCDTRSR registers respectively, see Section 4.5, “Global Channel Registers.”. The host CPU must write 1 to clear a status bit.

NOTE

eTPU A channels [0:2,12:15,28:29] and eTPU B channels [0:3,12:15,28:31] are connected to the DMA. The data transfer request lines that are not connected to the DMA controller are left disconnected and don’t generate interrupt requests, even if their request status bits assert in registers ETPUCDTRSR and ETPUCxSCR

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	CIS	CIOS	0	0	0	0	0	0	DTRS	DTROS	0	0	0	0	0	0
W	CIC	CIOC							DTRC	DTROC						
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr Channel_Register_Base + 0x4

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	IPS	OPS	OBE	0	0	0	0	0	0	0	0	0	0	0	FM	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reg Addr Channel_Register_Base + 0x4

Figure 4-18. ETPUCxSCR Register

Table 4-29. ETPUCxSCR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	0	CIS	Channel Interrupt Status. 1 channel has a pending interrupt to the host CPU. 0 channel has no pending interrupt to the host CPU.
W	0	CIC	Channel Interrupt Clear. 1 clear interrupt status bit. 0 keep interrupt status bit unaltered. These bits are mirrored in ETPUCISR, see Section 4.5.1, “eTPU Channel Interrupt Status Register (ETPUCISR).”. See also Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.”
R	1	CIOS	Channel Interrupt Overflow Status. 1 interrupt overflow asserted for this channel 0 interrupt overflow negated for this channel
W	1	CIOC	Channel Interrupt Overflow Clear. 1 clear status bit. 0 keep status bit unaltered. These bits are mirrored in ETPUCIOSR, see Section 4.5.3, “eTPU Channel Interrupt Overflow Status Register (ETPUCIOSR).”. See also Section 5.2.2.2, “Interrupt and Data Transfer Request Overflow.”
—	2 – 7	—	Writes do not affect bit values. Reads return 0.
R	8	DTRS	Data Transfer Request Status. 1 Channel has a pending data transfer request. 0 Channel has no pending data transfer request.
W	8	DTRC	Data Transfer Request Clear. 1 clear status bit. 0 keep status bit unaltered These bits are mirrored in ETPUCISR, see Section 4.5.2, “eTPU Channel Data Transfer Request Status Register (ETPUCDTRSR)). See also Section 5.9.3.10, “Channel Interrupt and Data Transfer Requests.

Table 4-29. ETPUCxSCR Bit Field Descriptions

Read/Write	Bits	Name	Description
R	9	DTROS	Data Transfer Request Overflow Status. 1 data transfer request overflow asserted for this channel 0 data transfer request overflow negated for this channel
W	9	DTROC	Data Transfer Request Overflow Clear. 1 clear status bit. 0 keep status bit unaltered. These bits are mirrored in ETPUCDTROSR, see Section 4.5.4, "eTPU Channel Data Transfer Request Overflow Status Register (ETPUCDTROSR).". See also Section 5.2.2.2, "Interrupt and Data Transfer Request Overflow."
—	10 – 15	—	Writes do not affect bit values. Reads return 0.
R	16	IPS	Channel Input Pin State. This bit shows the current value of the filtered channel input signal state
R	17	OPS	Channel Output Pin State. This bit shows the current value driven in the channel output signal, including the effect of the external output disable feature, see Section 2.2.3, "Channel Output Disable Signals." If the channel input and output signals are connected to the same pad, OPS reflects the value driven to the pad (if OBE=1). This is not necessarily the actual pad value, which drives the value in the bit IPS.
R/W	18	OBE	Output Buffer Enable reflects the state of the OBE signal. It's controlled by microcode.
—	19 – 29	—	Writes do not affect bit values. Reads return 0.
R/W	30 – 31	FM[1:0]	Channel Function Mode[1]. Each function may use this field for specific configuration. These bits can be tested by microengine code, see Section 5.9.4.2.3, "Conditional/Unconditional Branch."

1. These bits are equivalent to the TPU/TPU2/TPU3 host sequence (HSQ) bits.

4.6.3 eTPU Channel x Host Service Request Register (ETPUCxHSRR)

ETPUCxHSRR is used by the host to issue service requests to the channel.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	Channel_Register_Base + 0x8															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	0	0	0	0	0	HSR		
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reg Addr	Channel_Register_Base + 0x8															

Figure 4-19. ETPUCxHSRR Register

Table 4-30. ETPUCxHSRR Bit Field Descriptions

Read/Write	Bits	Name	Description
—	0 – 28	—	Writes do not affect bit values. Reads return 0.
R/W	29 – 31	HSR[2:0]	Host Service Request. This field is used by the host CPU to request service to the channel, see Section 5.2.5, “Host Service Requests.” • HSR = 000: no host service request pending • HSR > 000: function-dependent host service request pending. HSR value turns to 000 automatically at the end of microengine service for that channel. The host should write HSR>0 only when HSR=0. Writing HSR=000 withdraws a pending request if scheduler did not begin to resolve the entry point yet, but it does not abort the service thread from that point on. For more details, see Section 7.2, “Entry Points,” and Section 5.2.5, “Host Service Requests.”

Chapter 5
Host Interface

5.1 System Configuration

System configuration registers are described in Section 4.2, “System Configuration Registers.” A specification for the initial configuration sequence is found in Section 12.1, “Configuration Sequence.”

5.2 Interrupts and Data Transfer Requests

5.2.1 Interrupt Types and Sources

Each of the eTPU channels can be a source of a channel interrupt request. eTPU A channels [0:2,12:15,28:29] and eTPU B channels [0:3,12:15,28:31] can be a source of a data transfer request. channel interrupts are targeted to the host CPU. Data transfer requests are targeted to the data transfer module, DMA in the MPC5554.

NOTE

The eTPU channels’ data transfer request lines that are not connected to the DMA controller are left disconnected and don’t generate interrupt requests, even if their request status bits assert in registers ETPUCDTRSR and ETPUCxSCR.

Interrupt and data transfer registers are used by the host to enable interrupts and data transfer requests, indicate their status and service them. Interrupt and data transfer requests have the same sets of registers and external signals, and are handled in the same way.

NOTE

Interrupt and data transfer requests can be cleared even when engines are in internal stop mode, through the global channel registers.

Channel interrupts and data transfer requests can only be issued by eTPU microcode, through one of the channel control instruction fields.

Both channel interrupt and data transfer requests can be individually enabled for each channel.

The eTPU interrupt and data transfer registers are mirrored in two organizations: grouped by channel and grouped by type (interrupt status, interrupt enable, data transfer status, data transfer enable). This allows either “channel-oriented” or “bundled channel” host interrupt service schemes, or a combination of them. For a detailed description, refer to Section 4.4, “Channel Registers Layout,” and Section 4.5, “Global Channel Registers.”

The eTPU can also assert a global exception interrupt indicating a global illegal state. There are three possible sources for a global exception:

- The execution of an illegal instruction by the microengine, see Section 9.6, “Illegal Instructions.” This global exception source is flagged by the bits ILFA and ILFB in register ETPUMCR.
- An SCM signature mismatch detected by the multiple input signature calculator (MISC). See Section 10.3.1, “SCM Test for MISC (Multiple Input Signature Calculator).” This source is flagged by the SCMMISF bit in the ETPUMCR register if enabled.
- A microcode request, through microinstruction field CIRC, see Section 9.4.10, “Channel Interrupt and Data Transfer Requests.” This global exception source is flagged by either the MGEA bit (engine A) or the MGEB bit (engine B) in the ETPUMCR register. The microcode has the ability to write an error code in the SPRAM to indicate the cause of the exception.

Global exceptions cannot be directly disabled within eTPU, except by disabling its sources (MISC and microcode), and they are cleared by the host by writing 1 to the GEC bit in ETPUMCR. Clearing global exception clears all global exception source status bits (ILFA, ILFB, SCMMISF, MGEA, MGEB). The assertion of global exception by one of the sources above does not prevent the others from asserting it too, so any number may be set at one time.

NOTE

There can be a race between the clear of a global exception and occurrence of a new set condition, such that the set happens just before the clear and cannot be sensed by the host. Therefore, global exceptions cannot be used as a normal interrupt source; it should only be used for emergency procedures.

5.2.2 Interrupt and Data Transfer Request Overflow

If a channel interrupt was issued, its status bit is still set, and microcode issues another channel interrupt, the channel interrupt overflow status (CIOS) bit is set for that channel. The CIOS bit in the channel status register ETPUCxSCR allows the host to check the interrupt overflow status, Section 4.6.2, “eTPU Channel x Status Control Register

(ETPUCxSCR)," The CIOS bit is mirrored in register ETPUCIOSR, Section 4.5.3, "eTPU Channel Interrupt Overflow Status Register (ETPUCIOSR)." Interrupt overflow status is not cleared automatically when interrupt status is cleared; the CIOC bit must be set to clear the CIOS. The same mechanism and respective registers (ETPUCDTROSR) are available for data transfer requests.

A global exception has no overflow status.

5.3 Parameter Access

5.3.1 Parameter Access Widths

From the host side the SPRAM address space is mapped in bytes, and each 32-bit parameter occupies 4 contiguous, aligned bytes. The host can read/write the SPRAM by 8-, 16-, or 32-bit accesses in aligned addresses.

In 32-bit access, The host can access all 32 bits or only the lower 24 bits with an automatic sign extension, see Section 5.3.4, "Parameter Sign Extension Area."

5.3.2 Parameter Addresses and Endianess

To access parameter number *xxx*, eTPU microengine(s) would select address xxx. The host would add *(xxx*4)* to the SPRAM base address to access the same parameter. For example, parameter 0x101 is seen by the host as *(SPRAM base address +0x404)*. An example of the SPRAM memory map is shown in Figure 5-1. The host can access the SPRAM with a 32-bit-wide bus cycle to a four-byte aligned address, 16-bit-wide bus cycle to a two-byte aligned address, or 8-bit wide bus cycle to any byte address. The address of the 24-bit parameters and the most significant byte depends on the endianness of the MCU. For more details, see Section 12.6, "Endianness."

5.3.3 Parameter Concurrency

Host accesses to parameters may occur in parallel with eTPU microengine accesses. Readings taken from a group of parameters while they are being simultaneously updated may lack coherency. The eTPU provides mechanisms to ensure parameter coherency in accesses from both host side and microengine side, including the use of a coherent dual-parameter transfer mechanism, described in detail in Section 5.7, "Parameter Sharing and Coherency."

5.3.4 Parameter Sign Extension Area

The SPRAM address space to the host is mirrored in a parameter sign extension (PSE) area, see Chapter 3, "Memory Map." Accesses from the host to the PSE area differ from accesses to the standard SPRAM address space as follows:

- Writes: the most significant byte of the parameters is not written and the SPRAM retains the old byte value, regardless of the host access size.
- Reads: the most significant bit of the 24-bit parameter (that is, the msb of the second most significant 32-bit parameter byte) is repeated in the 8 most significant bits of the read value on all 32-bit reads and most significant 16- and 8-bit reads.

The parameters written in the standard SPRAM address space are read from the PSE area with the same offsets, and vice-versa. Refer to Table 12-9 for a reference of the address offsets in big and little endian machines.

Having a PSE area relieves the host from extending the signal of 24-bit eTPU parameters before calculations, and from read-modify-write accesses to modify 24-bit parameters at the SPRAM.

5.4 SPRAM Organization

The SPRAM internal partition for channel allocation is dynamic and programmed in the channel registers, see Section 4.6.1, “eTPU Channel x Configuration Register (ETPUCxCR).”

The host application is responsible for allocating a different parameter base address to each channel during the initial eTPU configuration, and to allocate enough parameters for the selected function, with no unintentional overlapping between parameters of different functions.

Besides channel parameters, global areas may have to be allocated for parameters that are shared by more than one channel, in one or both engines. Also, temporary parameter areas should be reserved to be used by the coherent parameter transfer mechanisms described in Section 5.7, “Parameter Sharing and Coherency,” if necessary.

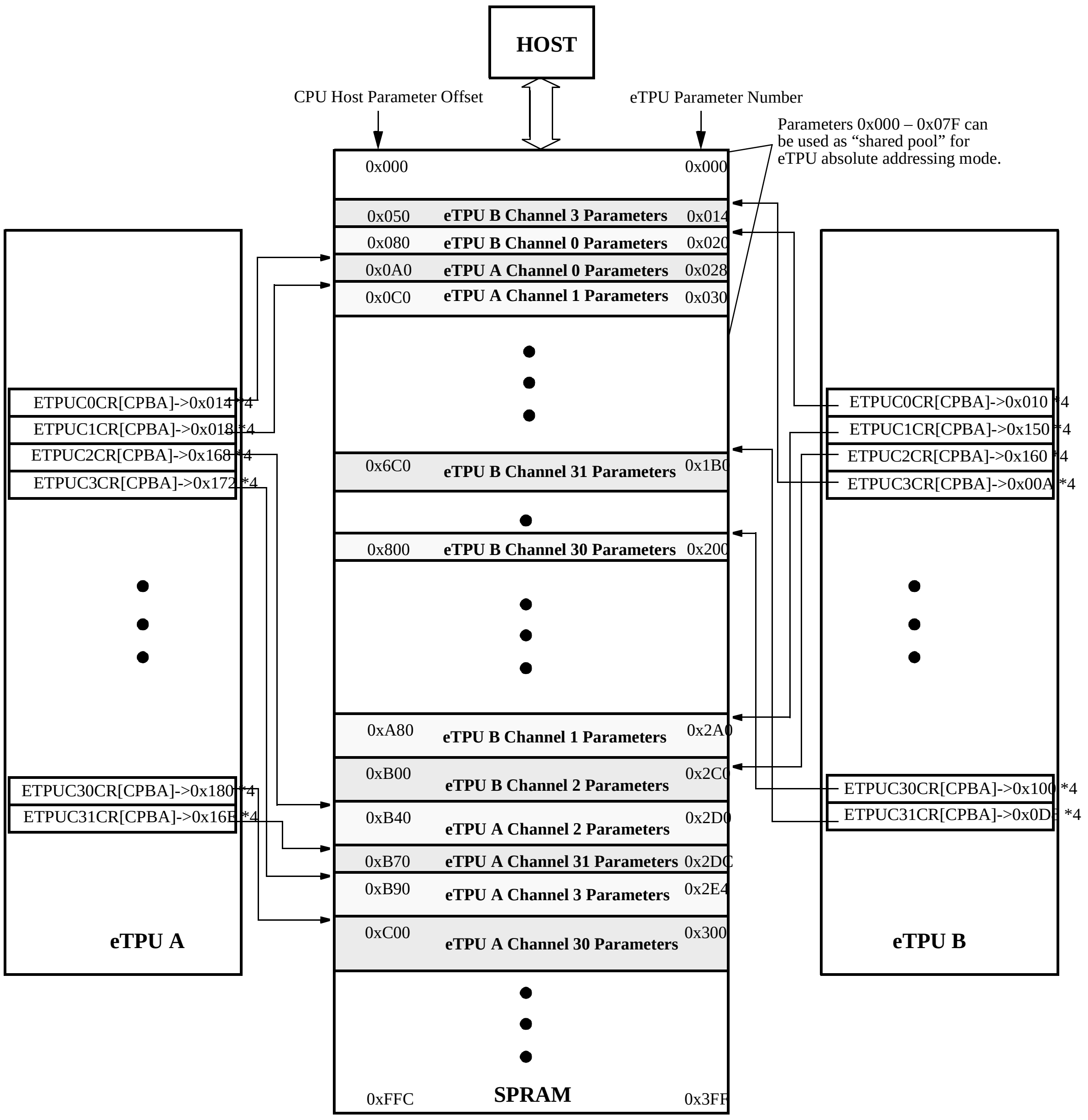

HOST
CPU Host Parameter Offset
eTPU Parameter Number
Parameters 0x000 – 0x07F can be used as "shared pool" for eTPU absolute addressing mode.
0x000 0x000
0x050 eTPU B Channel 3 Parameters 0x014
0x080 eTPU B Channel 0 Parameters 0x020
0x0A0 eTPU A Channel 0 Parameters 0x028
0x0C0 eTPU A Channel 1 Parameters 0x030
0x6C0 eTPU B Channel 31 Parameters 0x1B0
0x800 eTPU B Channel 30 Parameters 0x200
0xA80 eTPU B Channel 1 Parameters 0x2A0
0xB00 eTPU B Channel 2 Parameters 0x2C0
0xB40 eTPU A Channel 2 Parameters 0x2D0
0xB70 eTPU A Channel 31 Parameters 0x2DC
0xB90 eTPU A Channel 3 Parameters 0x2E4
0xC00 eTPU A Channel 30 Parameters 0x300
0xFFC SPRAM 0x3FF
ETPUC0CR[CPBA]->0x014 *4
ETPUC1CR[CPBA]->0x018 *4
ETPUC2CR[CPBA]->0x168 *4
ETPUC3CR[CPBA]->0x172 *4
ETPUC30CR[CPBA]->0x180 *4
ETPUC31CR[CPBA]->0x16E *4
eTPU A
ETPUC0CR[CPBA]->0x010 *4
ETPUC1CR[CPBA]->0x150 *4
ETPUC2CR[CPBA]->0x160 *4
ETPUC3CR[CPBA]->0x00A *4
ETPUC30CR[CPBA]->0x100 *4
ETPUC31CR[CPBA]->0x0D *4
eTPU B

Figure 5-1. SPRAM Organization Example

A single-engine eTPU or dual eTPU system may require fewer parameters than the maximum number provided by the SPRAM. Since the SPRAM partition is flexible, there is no limitation of fixed channel addresses, and the reduced array can be fully utilized.

5.5 Host Service Requests

The host CPU can request immediate service from a channel by writing a non-zero value to the host service request register field (HSR), see Section 4.6.3, “eTPU Channel x Host Service Request Register (ETPUCxHSRR).” There is one HSR field for each channel, so that writing to it generates a service request to the respective channel only. A zero value in HSR means no host service request is pending for the channel.

The meaning of a non-zero HSR value is defined by software. The HSR bits are part of the conditions which select the function entry point, and cannot be tested by microcode.

If the host writes HSR=000 when a thread for the same channel is already running, the thread runs until the end and is not aborted. If the host writes HSR>000 when a thread for the same channel is already running a thread started by a host service request, HSR value resets at the end of the thread, and no new HSR will be pending. If HSR is written before its value is resolved by the scheduler during TST, the entry point will obey the new HSR value. If this new HSR value is 000, no service thread is executed for the HSR, but other service requests may be resolved.

The scheduling of HSRs is completely asynchronous with host accesses, and there is no race-free manner to change an HSR value before service thread execution, so generally the safe way is: write HSR>0 only when HSR=0. Error recovery or emergency host procedures may require one to the safely abort service and reset channel state when an HSR is already pending or executing. In fact, normal operations could cause HSR interference, such as when the host initiates an service request, in the background, and then immediately issues another in an asynchronous foreground routine. In these cases, the procedure below should be followed:

1. Disable the channel, writing 00 to CPR, the channel priority field, in register ETPUCxCR. That will prevent any pending HSR from being serviced.
2. Check if the channel is currently being serviced, reading its service status bit in register ETPUCSSR. If it is, wait for the time necessary to finish the service pending, or check again until HSR == 0, or channel service bit in ETPUCSSR is cleared.
3. Write HSR with the error recover value. This value should, possibly combined with other host-defined flags in SPRAM or FM bits, initiate a channel reset or error recovery procedure.
4. Re-enable the channel, writing CPR value > 0 in register ETPUCxCR.

5.6 SCM Access

Only the host can access SCM as data. The SCM is implemented as RAM on the MPC5554. This characteristics for host accesses to the SCM as shown below.

5.6.1 SCM RAM Implementations

When SCM is implemented as RAM, the host may read or write to SCM by setting ETPUMCR bit VIS=1. If VIS=0 and the host tries to access SCM space, a bus error is issued. Both engines must be stopped or halted to set VIS=1.

Only 32-bit aligned writes are allowed to SCM from the host. Write accesses of other sizes store unpredictable values into SCM.

NOTE

It is necessary to turn VIS bit on to set software breakpoints. This is because a software breakpoint is actually a change of a microcode instruction. See Section 10.2.5, “Software Breakpoints,” for more details.

5.6.2 SCM Low Power

SCM turns off its internal clocks and stops MISC when both engines are stopped (ETPUECR bit STF asserted), VIS=0 at ETPUMCR, and MISC is not enabled (SCMMISEN=0). The SCM clocks are automatically turned on (along with MISC if SCMMISEN=1) if either one of the STF bits is negated or VIS is set to 1. Note that SCM cannot enter low power mode and MISC does not run if VIS=1: ETPUECR[STOP] are write-protected in this case.

SCM clocks are not turned off if any of the engines are not stopped, even if they are both halted.

5.7 Parameter Sharing and Coherency

SPRAM can be concurrently accessed by the host CPU and Microengine(s) (two in a Dual eTPU engine system). In general, the actual time of accessing parameters by the various engines is not guaranteed. Particularly the timing of a read with respect to a write of the same group is not guaranteed. These factors may lead to a lack of internal consistency if two or more related parameters are read when only part of them is updated.

The eTPU provides mechanisms to guarantee parameter coherency. The most generic mechanisms for host-eTPU coherency, suitable for any number of parameters, are:

- the use of transfer service thread mechanism.
- the mailbox (or “software semaphore”) mechanism.

These mechanisms, described in Section 12.3, “Multiple Parameter Coherency Methods,” use microcode to transfer parameters from temporary buffers in SPRAM to their definitive locations (or vice-versa). Though these methods maintain coherency, they have the disadvantage of wasting processing resources.

The eTPU also provides a coherent dual-parameter controller (CDC) mechanism. It is used by the host to coherently transfer pairs of parameters from/to a parameter buffer located on SPRAM to/from the locations on SPRAM where parameters are accessed directly by the channels. Coherency is guaranteed by SPRAM access arbitration. Although limited to two parameters only, it has lower latency and wastes no microengine resources, see Note: . CDC usage is described in Section 5.7.3, “Coherent Dual-parameter Controller (CDC).”

NOTE

A microengine access to the SPRAM at the moment the CDC is performing a transfer may suffer a maximum of two wait-states.

For parameters shared by both engines, the eTPU provides Hardware Semaphores. Coherency is assured given the semaphores are used to prevent concurrent access to the changing parameters. A Microengine can request semaphores using specific microinstructions.

Neither the host nor CDC have access to the hardware semaphores, but they can be combined with microcode transfer mechanisms if the host must coherently access parameters which are also shared by both engines.

In order to ensure coherent access to a group of parameters by two or more contenders, each contender must have atomic access to the shared parameters. Atomicity conditions are discussed in Section 5.7.1, “Host Side Atomic Access,” and Section 5.7.2, “Microengine Side Atomic Accesses.”

5.7.1 Host Side Atomic Access

Host side atomic accesses can be achieved by either of following ways:

- for one parameter, the SPRAM should be accessed by 32-bit-wide data transfers to ensure coherency between different fields in a 32-bit parameter, for instance.
- for two parameters only, using the Coherent Dual Parameter Controller.
- indirectly, for any number of parameters, by requesting microcode to coherently access SPRAM in its behalf. The host side atomicity problem becomes, then, a microengine side atomicity problem. Some methods that use this approach to achieve coherency are described in Section 12.3, “Multiple Parameter Coherency Methods.”

5.7.2 Microengine Side Atomic Accesses

5.7.2.1 Microengine Single Parameter Atomicity

SPRAM should be accessed by 32-bit-wide data transfers to ensure atomicity for 32-bit parameters and 24-bit accesses for 24-bit parameters. This applies either to host-microengine coherency or microengine-microengine coherency in a dual eTPU engine system.

5.7.2.2 Microengine Dual Parameter Atomicity

The microengine has the ability to access two parameters coherently in back-to-back accesses, at random addresses: once it accesses SPRAM, it has priority over the host for another access in the next microcycle, see Section 5.7.5, “SPRAM Arbitration.” Note that it applies only to microengine-host coherency. For microengine-microengine coherency in a dual eTPU engine system, one must use Hardware Semaphores, see Section 5.7.4, “Hardware Semaphores.”

Microengine dual back-to-back accesses are guaranteed to be atomic in relation to host accesses or coherent dual-parameter controller, regardless of semaphore usage: host or CDC accesses cannot break-up a back-to-back microengine access, neither microengine can break a CDC transfer, due to the SPRAM arbitration mechanism described in Section 5.7.5, “SPRAM Arbitration.”

Atomicity is not guaranteed if microengine enters halt state in the middle of a back-to-back access: the host can access SPRAM while microengine is halted in the middle of a back-to-back access.

5.7.2.3 Microengine Side Multiple Atomicity

Hardware semaphores must be used for microengine-microengine coherency (more than 1 parameter) since two or more accesses from one Microengine are not atomic with respect to the other.

For multiple microengine-host coherency, the software methods described in Section 12.3, “Multiple Parameter Coherency Methods,” or similar ones, must be used. Some of these methods rely on the fact that parameter access of a thread is atomic in relation to another thread in the same engine, since a thread cannot be suspended (preempted).

For 1 parameter coherent access, or dual parameter coherency between only one Microengine and host, the alternatives shown in previous sections apply.

5.7.3 Coherent Dual-parameter Controller (CDC)

Dual-parameter coherency is supported by a coherent dual-parameter controller (CDC), which contends with microengine for SPRAM access. The CDC atomically transfers, upon the host’s command, two parameters from one area of the SPRAM to another, as illustrated in Figure 5-2.

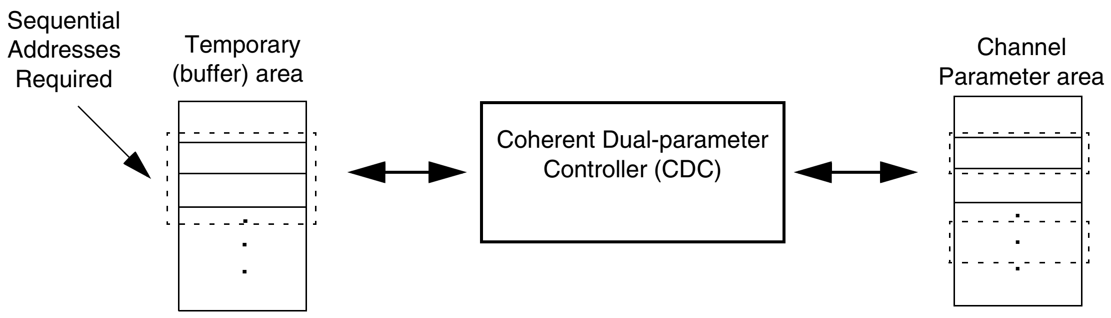


Figure 5-2. CDC Block Diagram

One area is a temporary (buffer) area, where the two parameters are directly read or written by the host. This temporary area has to begin in an SPRAM address multiple of 2 words, and the two parameters must be sequential. The other area is the channel parameter area where the microcode normally accesses the parameters, usually with the channel relative address mode, see Section 9.2.1, “SPRAM Addressing Modes.” In this area, the parameters transferred by CDC don’t have to be sequential. A transfer from the temporary area to the channel area, when the host sends data to the channel, is called a write transfer. Inversely, in a read transfer the parameters are copied from the channel area to the temporary area (channel to host).

Coherency is guaranteed by the SPRAM access contention rules implemented in the SPRAM arbiter, see Section 5.7.5, “SPRAM Arbitration.” CDC transfers are coherent in respect to the two engines, so the target parameters in the channel area may be shared by channels on both of the engines. During CDC operation, the host may suffer from 4 to 11 system clocks wait states, see Note: , and the microengine(s) may suffer up to 2 microcycle wait-states, see Note: . CDC accesses are atomic with respect to microengine(s) accesses to the SPRAM. CDC may also suffer up to 3 system clock wait-states from SPRAM arbiter, so that it does not break atomic back-to-back accesses from microengine(s). The CDC may not preempt TST preload accesses. The host CPU can initiate back-to-back CDC transfers, i.e. the parameter bus does not need to be idle for any period between two transfers.

NOTE

The maximum number of host wait states on CDC occurs when both microengines overlap their TSTs, delayed 3 system clocks from each other.

NOTE

One microcycle takes two system clocks:. Microengines get wait-states in multiples of microcycles, while host and CDC wait-states are multiples of system clocks.

5.7.3.1 CDC Programming

The coherent dual-parameter controller register, see Section 4.2.2, “eTPU Coherent Dual-Parameter Controller Register (ETPUCDCR), is used to configure and initiate CDC transfers between the temporary area and channel parameter area. The host asserts the STS bit in order to start the data transfer. The CDC then contends for the SPRAM and starts the transfer. When the data transfer is complete, STS returns to 0. The host receives wait-states for writing STS=1 while the CDC contends for SPRAM and during the transfer. The write access ends when the CDC finishes the transfer. The host receives wait-states during a CDC transfer. If the host writes ETPUCDCR with STS=0 or does not write the STS byte, the CDC transfer does not occur.

CDC programming can be summarized as follows:

1. if it is a write transfer, i.e., from host to a channel, write the two parameters into a temporary area.
2. write ETPUCDCR with STS=1 and the remaining CDC programming parameters: parameter width (32 or 24 bits, field PWIDTH), transfer direction (read or write, field WR), temporary parameter area base address (field PBBASE), and the absolute addresses of the parameters to be transferred (concatenation of the fields {CTBASE, PARM0/1}*4). If it is a read transfer, i.e., from channel to host, read the two parameters from the temporary area into host memory/registers.
3. if it is a read transfer, i.e., from a channel to host, read the two parameters from the temporary area into Host memory/registers.

5.7.4 Hardware Semaphores

The eTPU provides Hardware Semaphores accessible by the Microengine only. It is the responsibility of the application to ensure proper use of the semaphores (i.e. agree upon a specific semaphore and use it properly, to ensure coherency).

The eTPU microinstruction set has support for locking and freeing the semaphores, described in, Section 9.2.7, “Semaphore Operations,” and this is the only way to access them.

There are four semaphores available, which reduces the amount of collisions by assigning unrelated data transfers to different semaphores. Semaphores are used for parameters which can be shared by channels in different engines, and for engine-to-engine synchronization. Semaphores are also the only way to ensure coherent access to parameters shared between the two microengines.

Attempting to lock one semaphore (even not successfully) frees the other locked by the same engine, ensuring one can lock just one semaphore at a time. That prevents deadlock conditions between the two engines.

A microcode END command or engine being in idle state (no thread executing) automatically releases any semaphores from one engine side. However, it is recommended to write microcode in a way which locks semaphores for the shortest required period, and frees them without waiting for the END command, to improve the performance of the other microengine. Semaphores are free after reset. An engine can only free a semaphore locked by itself.

Semaphore lock requests are always non-blocking, in the sense that they do not suspend the requester in case the semaphore is already locked. The semaphore status after the lock request, indicating if it was successfully locked or not, must be tested through the SMLCK microengine branch condition, see Section 8.5, "Branch Conditions."

5.7.5 SPRAM Arbitration

Up to four entities can access SPRAM:

- two microengines (in a dual eTPU engine system)
- the coherent dual-parameter controller (CDC)
- the host CPU (direct memory-mapped access)

The following rules specify the access priorities for contended access. First level arbitration keeps compatibility with the TPU3 dual parameter access atomicity, but only between the microengine and CDC.

1. Microengine accesses from the two eTPU engines are interleaved between each other, but not with host or CDC accesses.
2. The eTPU microengine(s) gives priority for SPRAM accesses to either the host CPU or the CDC under any of the following conditions:
   a) the microengine has completed accessing the second parameter in a back-to-back SPRAM access.

   NOTE

   If microengine tries to access the SPRAM in the following microcycles, the third and fourth consecutive accesses are considered the first and second of a new back-to-back dual access.

   b) the SPRAM was not accessed during the last microengine arbitration slot.

   NOTE

   The microengine access slot is between microengine's T4 and T2 edges, see Appendix A, "Microcycle Timing."

3. The eTPU microengine takes priority for SPRAM accesses under either of the following conditions:
    a) the host CPU or CDC has completed a data transfer during the last access arbitration slot for the engine, see Note: . Also, the host CPU does not hold a pending access against the other eTPU microengine.
    b) the microengine is arbitrating for the access of its second parameter in a back-to-back access, see Note: . All pairs of back-to-back parameter accesses are coherent with respect to host and CDC (not to the other microengine).

The direction (read or write) of any individual access by the host or microengine is irrelevant to the arbitration. The use of normal or PSE SPRAM area by the host is also irrelevant to the arbitration.

The first parameter preloading in a TST is considered first access by the arbiter, regardless of any access made at the END microinstruction of the previous thread, i.e.: the last access of a thread and the first preload are never considered a back-to-back access. On the other hand, the TST preload accesses are considered back-to-back and therefore atomic with respect to host or CDC.

NOTE

The Zero SPRAM operation, see Section 9.2.5, “Zero SPRAM Operation,” is considered an SPRAM access for arbitration.

5.8 Enhanced Channels

Enhanced Channels comprise hardware support for input signal detection (transition event) and output signal generation (match event). Each Channel is associated with one input and one output signal. Enhanced channel logic is configured with function microcode (and optionally angle mode logic) to implement channel I/O functionality.

The eTPU’s enhanced channels are capable of dual action, meaning that each channel logic can handle two events at different times and/or cause two separated actions; these actions and events can be mutually dependent (with the first either blocking or enabling the other), or both independent, depending on the programmed channel mode.

Each enhanced channel contains event logic containing two event register sets, each set supporting one input and/or output action, the pair implementing dual action support. Each event register set contains two 24 bit registers: match and capture. The match register holds the pending match value which is compared against one of the two time bases by an equal-only/greater-equal comparator. The capture register captures one of the two time bases as a result of a match or transition detection. Service requests are issued on particular combination of match and capture events, defined by the selected channel mode.

In the context of the eTPU channels, a match is a comparison between a time base value and a channel match register. If those two values are coincident, or the time base value is greater than the value of the match register, a match event occurs. Depending on the channel mode of operation and current state of the channel logic, the match event may be recognized, i.e., change the state of the channel, or be ignored. A match event recognized by the channel logic is called a match recognition. Depending on channel mode and current state, match recognitions can cause the channel to request service or other actions (such as window enabling), configuring a match service request.

The eTPU uses two kinds of comparators to assert a match event: an Equal comparator, in which both the match register and the value of the selected time base must match exactly, and a greater-equal comparator. The greater-equal comparator considers any time base value between the range [N – N+0x800000-1] as a valid match against the value of N in the match register, even when the value N+0x800000-1 wraps around the point of origin (0x0). Refer to Figure 5-3 for an explanation about the matching values on a greater-equal comparator.

The second source of events for the eTPU channel is a Transition detected at the corresponding channel's input signal. Two distinct Transition detections can be programmed individually for each channel, allowing recognition of several possible combinations of edge detection. It is also possible to check the sampled state of an input signal upon the occurrence of a match: the sampling of the expected value is treated as a transition, even if the input signal did not necessarily toggled at the time of the match, or at any time at all.

Like match events, transition events may or not be recognized by the channel logic. When they are, a transition detection occurs. As well as match recognitions, transition detections can issue a channel service request, depending on channel mode and current state.

Transition detections and match recognitions are sometimes simply called Transitions and matches throughout this document, for short.

Input and output signals can be processed separately by the channel logic and microcode, and can also be combined such that matches and Transitions are used to cause output signal actions. The output signals can also be directly controlled by microcode. Many event combinations are allowed for a channel, given the possibility of configuring pairs of matches and transitions for the dual action logic, where each event is able to block or enable the next one. There is a full set of channel modes described in Section 5.8.4, "Channel Modes," exploring all the capabilities mentioned here.

Each channel has its own set of registers and flags. They are selected, and made accessible to the Microengine, according to the value written into the microengine CHAN Register that points to the desired channel. Every time the CHAN register is written, even if with the same previous value, a channel is selected and its flags and registers are updated. For further detail, see Section 5.8.1.3.1, "Channel Selection Register (CHAN)."

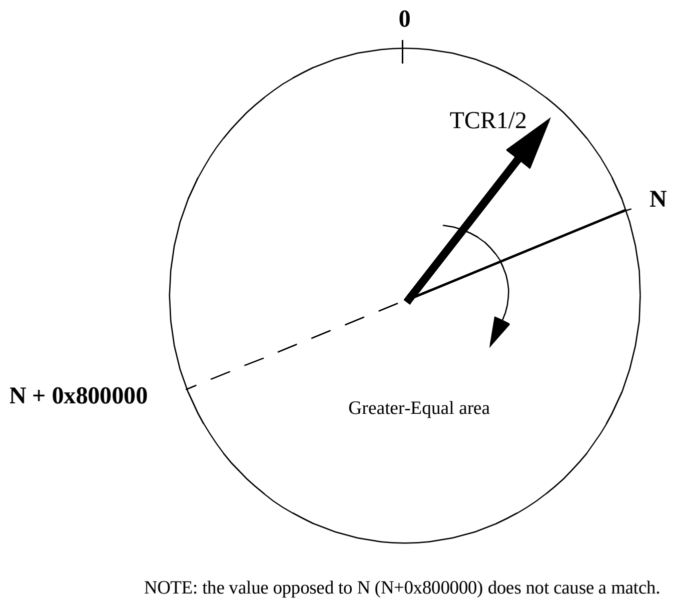

0
TCR1/2
N
N + 0x800000
Greater-Equal area
NOTE: the value opposed to N (N+0x800000) does not cause a match.

Figure 5-3. Greater-Equal Comparator

Beyond the request of services due to the signal and timing internal to each channel, one eTPU channel microcode can explicitly request service from another channel through the microengine LINK Register. A microcode write to the LINK Register asserts a service request to the channel whose number matches the contents of LINK. Refer to Section 5.8.5, “Channel Link,” for a complete description of this mechanism.

Service requests originating in the eTPU enhanced channels (either time base match, input signal transition, or link service request) result in a call to the corresponding channel service routine, which is a sequence of microinstructions called a thread. For further detail, refer to Chapter 7, “Functions and Threads.”

In addition to event logic, each channel has an enhanced digital filter which eliminates spurious glitches on input pin signal. See Section 5.8.6, “Enhanced Digital Filter (EDF),” for more information.

A high level diagram of channel logic and registers is shown in Figure 5-4.

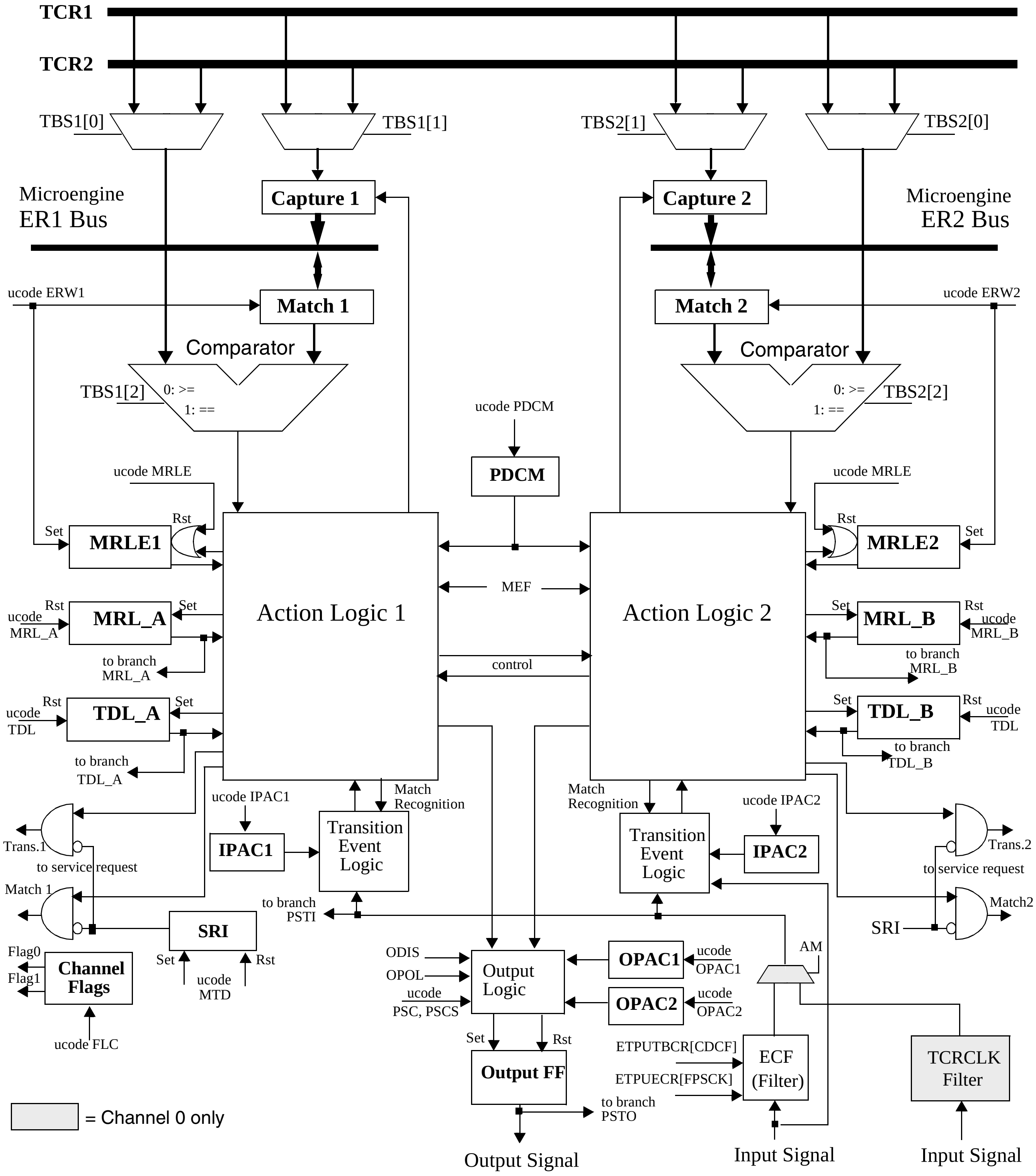


Figure 5-4. Channel Logic Block Diagram

5.8.1 Channel Registers and Flags

Channel configuration and control registers can be divided in the following groups:

- Host configuration and control registers, which define channel function and parameter allocation in SPRAM, input signal filtering, manage host interrupts, and are used for host service requests; they can only be accessed by the host, except for the function mode bits which can be written by the host and tested by the microengine.
- Event registers, which can only be accessed by eTPU microengine, through dedicated channel control microinstruction operations, see Section 5.5.3, "Channel Control and Configuration Microoperations." These registers are directly used to implement channel functionality, and include channel event status latches which can be directly tested by microengine branch instructions.
- Pin control registers, which basically define pin state and transition polarity (but not input signal filtering); they are accessible only by dedicated channel control microinstruction operations.
- Link registers, which implement the channel link mechanism that allows one channel to request service to another one; they are accessible only by microinstruction operations.
- General channel registers: CHAN, SRI, Flag0/1.

Most of the aforementioned registers are channel exclusive, i.e., there is one copy of them for each channel. Microcode can access registers from only one channel at a time. With exception of link register and function mode, the Channel Selection (CHAN) register defines the channel whose registers are being accessed. CHAN may be set by the thread entry process or is accessible by microcode. For more details, see Section 5.8.1.3.1, "Channel Selection Register (CHAN)."

The service request inhibit (SRI) register controls the generation of service requests on matches and transitions, also affecting channel logic behavior. For a full description see Section 5.8.1.3.3, "Match/Transition Service Request Inhibit Latch (SRI)." Flag0/1 are used to select channel service threads based on channel software state. See Section 5.8.1.3.4, "Channel 'State Resolution' Flags (Flag1), (Flag0)," for more details.

Host configuration and control registers are described in Section 4.6, "Channel Configuration and Control Registers."

TCR1/2 is common to all channels in a microengine, but the selection of which time base is used for a specific match or capture is individual to the channel (and selected in the channel hardware). For more detail, see Section 5.9, "Time Bases."

Link registers are described in Section 5.8.5, "Channel Link."

5.8.1.1 Event Registers (ER)

Each channel contains two identical event register sets, named ER1 and ER2, corresponding to the two actions supported. Each event register set includes:

- a 24-bit match register (Match1 or Match2), which holds a match value. This value is compared against the selected match time base (TCR1 or TCR2).
- a 24-bit Capture register (Capture1 or Capture2), which samples the selected capture time base (TCR1 or TCR2)
- a time base selection register (TBS1 or TBS2)
- a match recognition status flag (or latch) (MRL_A or MRL_B)

  For more information, see Section 5.8.2.1, "Match Recognition Latches (MRL_A/B)."
- a match recognition enable latch (MRLE1 or MRLE2)

  For more information, See Section 5.8.2.3, "Match Recognition Latch Enable (MRLE1/2)."
- a Transition Detect flag (or latch) (TDL_A or TDL_B)

  For more information, see Section 5.8.3.1, "Transition Detect Latches (TDL_A/B)."

ER1 and ER2 are associated with the first and second events in double action modes, always in that order for transition detections, but not necessarily for match recognitions. The order of match events associated with ER1 and ER2 depends on the programmed channel mode, the Match1 and Match2 values, and the timebases selected by TBS1 and TBS2.

These registers of ER1 and ER2 are directly or indirectly accessed by the microcode. TBS1 and TBS2 registers are defined in Section 5.8.1.1.3, "Time Base Selection Registers (TBS1) and (TBS2)." The other registers composing ER1 and ER2 are explained in Section 5.8.2, "Match Recognition," and Section 5.8.3, "Transition Detection and Time Base Capture."

Access to the event registers is qualified by the channel currently selected by the microengine (i.e., the channel value currently in the CHAN register). During the channel transition period (automatic CHAN assignment), or whenever CHAN is written by microcode, Capture values of the new selected channel are sampled into Microengine registers ERT_A and ERT_B, therefore becoming visible to the microcode. At the same time, updated values of MRL_A, MRL_B, TDL_A and TDL_B are sampled into the branch logic, making the register values and the flags coherent with respect to each other and with the thread selected by the scheduler, see Note: .

NOTE

The function mode bits are also sampled from the host interface on time slot transition, so that they remain constant to microengine even when the host changes them.

NOTE

The thread selected is determined by the entry point which, in turn, is determined partially by the channel latches. See Section 7.2.2, "Entry Point Address Generation."

During service, the microcode can access updated values of the event registers of any channel by writing the channel number to CHAN. Writing CHAN with the same value (CHAN := CHAN) updates ERT_A and ERT_B with the new captured values, the branch logic with updated MRL_A/B and TDL_A/B flags. Writing CHAN with a different value does the same with the values from the newly selected channel.

Match values are also accessed through ERT_A and ERT_B microengine registers, which are copied to/from the channel Match1 and Match2 registers by specific microinstruction operations.

Microcode writes to the flags and selections (MRL_A/B, TDL_A/B and TBS1/2) are immediately effective to the channel.

NOTE

Microcode may clear, but not set MRL_A, MRL_B, TDL_A, or TDL_B.

The MRL_A/B and TDL_A/B branch conditions are also immediately reset when their corresponding flags are reset by microcode. Match registers are indirectly written by microcode through ERT_A/B. MRLE1/2 is unconditionally asserted when respective match register is updated from ERT_A/B, and its negation by microcode is immediate. MRLE can also be negated by setting MRL.

Table 5-1 summarizes event registers accesses.

Table 5-1. Event Registers Microcode Accesses

Register	Access Type	Sampled from channel	Update to channel	Microcode fields[1]	Reset value[2]
Capture1, Capture2	read through ERT_A/B	to ERT_A/B on CHAN assignment	no	T2ABD	n.a.
Match1, Match2	read and write through ERT_A/B	to ERT_A/B by microcode	from ERT_A/B by microcode	ERW1, ERW2, T4ABS	n.a.
MRLE1, MRLE2	write to 0 (negate) directly; write to 1 (assert) upon Match1/2 update from ERT_A/B	no	immediate	MRLE, ERW1, ERW2	0, 0
TBS1, TBS2	write only	no	immediate	TBS1, TBS2	000, 000
MRL_A, MRL_B	flag test on branch, write to 0 (negate) only	on CHAN assignment	immediate	BCC (test) MRL_A, MRL_B (reset)	0, 0
TDL_A, TDL_B	flag test on branch, write to 0 (negate) only	on CHAN assignment	immediate	BCC (test) TDL (reset)	0, 0

[1] see section Chapter 9, “Microinstruction Set.”

[2] n.a. means that value of the register is undetermined after reset.

5.8.1.1.1 Match1 and Match2 Registers

Match1 and Match2 registers hold a match value to be compared against the selected channel time base. A match value can only be written into the match register by microcode, through ERT_A/B microengine registers, see Section 9.4.5, “Write Channel Match Registers.” Microcode can also read the match register as a special T4ABS source operation, when T4ABS=0101, and the source for T4ABS is selected from the second register set. In this operation, Match1/2 registers are copied into ERT_A/B registers, see Section 5.5.2.2, “Selecting Sources and Destination.” For more information on time base matches, see Section 5.8.2, “Match Recognition.”

5.8.1.1.2 Capture1 and Capture2 Registers

Capture1 and Capture2 registers capture the selected channel time base. Capture1/2 registers cannot be directly written or read by microcode. During the time slot transition (TST) or during CHAN assignment, Capture1/2 registers are copied into ERT_A/B microengine registers. For more information, see Section 5.8.3, “Transition Detection and Time Base Capture.”

5.8.1.1.3 Time Base Selection Registers (TBS1) and (TBS2)

TBS1/2 are 3-bit registers which have the following effect on channel configuration:

- Selection of the timebase (TCR1 or TCR2) to be compared against the match values in Match1 and/or Match2 registers.
- Selection of the timebase (TCR1 or TCR2) to be captured in the Capture1 and/or Capture2 registers by a match or transition detection event.
- Selection of comparator mode to be used with Match1 and Match2 registers: equal-only or greater-equal.

After reset TBS1/2 are 000. Table 5-2 shows values of TBS1 and TBS2 for configuration selection. Note that the time base selection for capture is independent of the time base selected for matches.

TBS bit	2	1	0
bit value	**Comparator selection**	**Capture time base selection**	**Match time base selection**
0	greater or equal	TCR1	TCR1
1	equal-only	TCR2	TCR2

Table 5-2. TBS1/2 Programming States

TBS1/2 are written through the microcode fields TBS1/2, see Section 5.5.3.2, “Comparator and Time Base Selection.”. Note that microcode field TBS1 is also used to control the OBE pin control register, see Section 5.8.1.2, “Pin Control Registers.”

5.8.1.2 Pin Control Registers

Pin control registers are replicated one per channel, accessed only by microcode and qualified by the CHAN register in the same way as event registers. Table 5-3 lists pin control registers, explained in following subsections, and their accesses.

Table 5-3. Pin Control Registers microcode accesses

Register	Access Type	Sampled from channel	Update to channel	Microcode fields[1]	Reset value
IPAC1, IPAC2	write only	no	immediate	IPAC1, IPAC2	000,000
OPAC1, OPAC2	write only	no	immediate	OPAC1, OPAC2	000,000
PSTI	flag test on branch	no	no	BCC	0
PSTO	flag test on branch, write	no	immediate	BCC (test) PSC, PSCS (set)	0
OBE	write only	no	immediate	TBS1 values 1000,1001	0 (negated)
PSS	flag test on branch	on CHAN assignment	no	BCC	0

[1] see Section 5.5, “Microinstruction Set.”

5.8.1.2.1 Input and Output Pin Action Control Registers (IPAC1), (IPAC2), (OPAC1), and (OPAC2)

These registers determine the input or output pin action which takes place due to match or transition events. Each field is three bits wide. After reset, the IPAC1/2 and OPAC1/2 registers are set to 000. IPAC1 and IPAC2 registers are mutually independent and have identical encoding, and so do OPAC1 and OPAC2. Table 5-4 shows IPAC and OPAC encoding. Output actions are triggered by matches, and can also be triggered by input actions (in special cases). Note that input actions are independent from the output actions.

The input actions specified by IPAC1/2 define the Transition events treated by channel logic. Although the name “transition” is generically used for the input actions, IPAC options 100 and 101 do not really detect transitions: they actually sample the state of the input signal at the occurrence of the corresponding match (Match1 used for IPAC1, Match2 used for IPAC2).

Table 5-4. IPAC1/2 and OPAC1/2 Encoding

value	IPAC meaning	OPAC meaning
000	Do not detect transitions	Do not change output signal
001	Detect rising edge only	Match[1] sets output signal high
010	Detect falling edge only	Match[1] sets output signal low
011	Detect either edge	Match[1] toggles output signal
100	Detect input signal = 0 on match[1]	Input action[2] sets output signal low
101	Detect input signal = 1 on match[1]	Input action[2] sets output signal high
110	Reserved	Input action[2] toggles output signal
111	N.A.[3]	N.A.[3]

[1] Match1 is used for IPAC1/OPAC1, and Match2 for IPAC2/OPAC2.

[2] An input action is the assertion of TDL_A in single action mode and TDL_B in double action mode.

[3] This value for fields IPAC1/2 and OPAC1/2 is neutral, meaning that IPAC/OPAC register values are not changed.

5.8.1.2.2 Output Pin Control Logic and Pin State Output Register (PSTO)

The output signal control logic uses OPAC1/2, the pre-defined channel mode (PDCM) and the microcode pin state control (PSC and PSCS) fields. It is responsible for setting the pin state output (PSTO) register to the specified logic value required by microcode, by input events, or by Match1 and/or Match2 events. The PSTO register stores the driven pin state determined by the pin control logic.PSTO register output also goes to the microengine branch logic, enabling branching on the driven pin state. PSTO is set to 0 on reset.

The PSC and PSCS microcode fields are used for setting the PSTO register to a fixed value, or to the value specified by the OPAC1 or OPAC2 microcode field, as shown in Table 5-5.

Table 5-5. PSC and PSCS encoding

PSC	Output Pin Action
00	Force pin state according to OPAC1 (PSCS=1) or OPAC2 (PSCS=0).
01	Force pin high.
10	Force pin low.
11	Do not change pin state.

For details refer to Section 5.5.3, "Channel Control and Configuration Microoperations."

When match recognitions or transition detections occur, the pin control logic sets PSTO value according to:

- the event number (Match1/Transition1 or Match2/Transition2)
- the contents of OPAC1/IPAC1 or OPAC2/IPAC2 registers
- the programmed channel mode

There are cases in which two match or transition events may occur at the same time, each of them trying to force a different pin action. The resolution of the selected match event which sets the value depends on the pre-defined channel mode (PDCM) register. For details refer to Section 5.8.4.30, "Match/Transition Pin Action Conflict Resolution."

5.8.1.2.3 Pin State Input and Pin Sampled State Registers (PSTI) and (PSS)

During the time slot transition period, or whenever the CHAN register is written by microcode, the PSTI filtered input signal is sampled into the branch logic and stored as PSS. Effectively, PSTI follows the pin whenever it changes, and PSS samples PSTI on CHAN write. The microcode can then branch on either the currently driven (PSTO) or input (PSTI) pin state, or on sampled pin state (PSS, which is stable as long as CHAN does not change).

5.8.1.3 General Channel Registers

These registers control other aspects of channel logic. Except for CHAN, they are unique per channel. Table 5-6 summarizes the registers and access options.

Table 5-6. General Channel registers microcode access

Register	Access Type	Sampled from channel	Update to channel	Microcode fields[1]	Reset value
CHAN	read/write	n.a.[2]	n.a.[2]	T4ABS, T4BBS, T2ABD	defaults to serviced channel at thread start
PDCM	write only	no	immediate	PDCM	1100 (sm_st)[3]
SRI	write only	no	immediate	MTD	1
Flag1,Flag0	write only (branch on test)	no	immediate	FLC	0, 0

[1] see Section 5.5, "Microinstruction Set."
[2] CHAN is common to all channels in the engine.
[3] See Section 5.8.1.3.2, "Pre-Defined Channel Mode (PDCM)."

5.8.1.3.1 Channel Selection Register (CHAN)

CHAN is the register that holds the number of the channel that qualifies the context of most channel registers, including pin control and ER accesses, and is common to all channels in a same engine.

When a thread starts to be executed, the contents of CHAN register are automatically updated on time slot transition to the number of the channel to be serviced. The serviced channel is constant during channel servicing, but the selected channel can be changed any time by microengine writing into CHAN register.

Some microinstructions are affected by the serviced channel instead of CHAN. These are:

- Conditional branch using LSR see Section 5.8.1.6, "LINK Register," or function mode in Section 4.6.2, "eTPU Channel x Status Control Register (ETPUCxSCR)."
- Negate channel flag LSR, interrupt CPU and data transfer request, see Section 5.5.3.10, "Channel Interrupt and Data Transfer Requests."

When CHAN register is written to, accesses are qualified by the new CHAN register value from the instruction following CHAN assignment on, except Capture 1/2 sampling into ERT_A/B and match register writing from ERT_A/B, see Section 5.5.6.5, "CHAN Assignment, Read Match and ERW1/2."

Writing CHAN (including with the same value, CHAN:= CHAN) updates ERT_A and ERT_B with the new captured values, and updates the branch logic with updated MRL_A/B and TDL_A/B flags.

Table 5-7 shows the commands, flags and registers selected by the CHAN register value.

Table 5-7. CHAN Selected Features

Feature Used	Selected by CHAN
channel-relative SPRAM access	YES
Branch using PSS,PSTI and PSTO channel flags.	YES
Branch using MRL_A/B, TDL_A/B[1]	YES
Branch on all other conditions[2]	no
ERT_A/B Value	YES
configure (selected) channel	YES
channel commands applied to: MRL_A/B, TDL_A/B, TBS1/2, IPAC1/2,OPAC1/2, PSC, PSCS, OBE, PDCM, MRLE	YES
channel command: set/reset SRI	YES
channel command: write to match registers (ERW1/2)[3]	YES[4]
channel command: read match registers into ERT_A/B[5]	YES[6]
clear LSR	no

[1] In the TPU, these conditions retained the old values.
[2] Refer to Section 5.5.4.2, “Branch Operations.”
[3] Assembler mnemonic write_mer1/2.
[4] If write match (ERW1/2) occurs at the same time of a CHAN assignment, the channel which is target of the write is the one selected by the new CHAN value, see Section 5.5.6.5, “CHAN Assignment, Read Match and ERW1/2.”
[5] Assembler mnemonic read_mer1/2.
[6] If read match occurs at the same time of a CHAN assignment, the match value is selected by the new CHAN value. See Section 5.5.6.5, “CHAN Assignment, Read Match and ERW1/2.”

5.8.1.3.2 Pre-Defined Channel Mode (PDCM)

PDCM determines the channel mode assigned to the channel. Channel mode defines much of the channel logic behavior, specially how matches block and enable transitions and vice-versa, as well as occurrence of time base captures and service requests based on matches and transitions. For a complete description see Section 5.8.4, “Channel Modes.”

PDCM is a 4-bit register set by the microcode field of the same name, see Section 5.5.3.9, “Predefined Channel Modes,” and cannot be read or tested in branch instructions. Table 5-8 relates the PDCM value with channel modes. The second column specifies the mnemonic used to reference the mode, introduced in Section 5.8.4.2, “Channel Modes Overview.” There is one PDCM for each channel, initialized with 1100 on reset.

Table 5-8. PDCM encoding

PDCM	Channel mode[1]
0000	em_b_st
0001	em_b_dt
0010	em_nb_st
0011	em_nb_dt
0100	m2_st
0101	m2_dt
0110	bm_st
0111	bm_dt
1000	m2_o_st
1001	m2_o_dt
1010	reserved
1011	reserved
1100	sm_st[2]
1101	sm_dt
1110	sm_st_e
1111	n.a.[3]

[1] for a description of channel modes refer to Section 5.8.4, "Channel Modes."

[2] this is the reset value, also compatible with TPU channel behavior.

[3] this value is used as a neutral (do not change) value in PDCM microinstruction field.

5.8.1.3.3 Match/Transition Service Request Inhibit Latch (SRI)

The SRI latch blocks channel service requests due to the assertion of MRL_A/B and/or TDL_A/B. The SRI does not affect recognition of link service requests or host service requests. The SRI also does not affect MRL_A/B or TDL_A/B microcode branch tests or entry table selection.

NOTE

In the TPU, SRI also blocked TDL and MDL branches.

The SRI is asserted during reset and is controlled by microcode field MTD.

To unburden the microengine, an asserted SRI, i.e. SRI = 1, mutes any match or capture channel service request to the microengine. Even with SRI=1, TDL_A/B and MRL_A/B can still be asserted, and the level specified by the OPAC (Output Pin Action Control) registers will be output to the pin on the appropriate match.

5.8.1.3.4 Channel ‘State Resolution’ Flags (Flag1), (Flag0)

Each channel has a pair of flags, simply called Flag0 and Flag1, that can be set/reset by microcode through microinstruction field FLC. These flags cannot be read or tested by microcode, but they are used to resolve the microcode entry point for the channel service, see Section 5.1.1, “Entry Points,”. Flag0 and Flag1 are, so, typically used for fast state resolution. FLC microinstruction field also allows Flag1,Flag0 to be copied from selected bits of P register high byte, which is also meant to be used to hold application state. Flag0 and Flag1 are both zero out of reset.

5.8.2 Match Recognition

The match operation is performed every microcycle by comparing the channel Match1 and Match2 registers against the value of the TCR bus specified for each match. TCR1 or TCR2 bus is selected according to TBS1 and TBS2 fields. Both results have effect on the next microcycle T2, see Appendix A, “Microcycle Timing.”

A Match1/2 event is qualified by a set of match enabling conditions to the match recognition registers MRL_A/B. To recognize the match and assert these registers, the following match enabling conditions are applicable

- For IPAC1/2=0xx, match enable flag (MEF), qualified by the channel currently being serviced must be asserted. Match1/2 is always enabled for IPAC1/2 =1xx, even during time slot transition (TST) and regardless of the state of the match enable flag (MEF). See Section 5.8.2.2, “Match Enable Flag (MEF),” for the conditions of MEF assertion.
- Match recognition latch enable 1/2 (MRLE1/2) is asserted. A match event recognition may only occur if its corresponding MRLE1/2 bit is set, which only happens upon a write to a channel match register by the microcode, copied from ERT_A/B. MRLE1/2 is negated when the respective match occurs or, in some double match channel modes, when a match for the other match register occurs. It ensures that the greater-equal comparison will not cause additional matches, see Note: .

NOTE

Microcode can also negate MRLE1/2.

- In selected modes, see Section 5.8.4, “Channel Modes,” the particular conditions of MRL and TDL flags of the other event, i.e:
  — MRL_A, TDL_A enable or block MRL_B;
  — MRL_B, TDL_B enable or block MRL_A.

- The respective MRL is negated.
- In selected modes, see Section 5.8.4, “Channel Modes,” the state of its respective TDL flag.

If the Match1 and/or Match2 conditions are met, the channel immediately forces the pin state if specified by the appropriate OPAC1/2 registers (Output Pin Action Control 1/2) and, in some cases, by IPAC1/2 registers. Refer to Section 5.8.1.2.1, “Input and Output Pin Action Control Registers (IPAC1), (IPAC2), (OPAC1), and (OPAC2).”

If both Match1 and Match2 events occur at the same time, with conflicting pin actions, the priority over the pin action is mode dependent. For further details on pin action resolution refer to Section 5.8.4.30, “Match/Transition Pin Action Conflict Resolution.”

5.8.2.1 Match Recognition Latches (MRL_A/B)

MRL_A/B indicate the recognition of a match event detected by the comparator. They are asserted on T2, see Appendix A, “Microcycle Timing.” Assertion of MRL_A/B issues a match service request in specific channel modes, depending on previous events and state of SRI. After reset MRL_A and MRL_B are both negated.

When MRL_A or MRL_B is asserted, it may change the output signal level according to the input and output pin action control registers, refer to Section 5.8.1.2.1, “Input and Output Pin Action Control Registers (IPAC1), (IPAC2), (OPAC1), and (OPAC2).” Assertion of MRL_A/B causes a capture of one or two time bases, according to the selected mode capturing scheme, see Section 5.8.3, “Transition Detection and Time Base Capture.”

A match recognition is self-blocking, regardless of channel mode: once MRL_A (MRL_B) has been asserted, it negates its associated MRLE1 (MRLE2) register, preventing future match recognitions, until the associated match register is rewritten by microcode. The microcode has to enable new matches by updating the new match value in the Match1 (Match2) register, see Note: . In addition, assertion of MRL_A/B can block its twin MRL_B/A, depending on the channel mode. In some double match blocking channel modes, Match1/2 event blocks the occurrence of Match2/1 in a “first win” scheme.

NOTE

Prior to this, the microcode should also negate MRL_A (MRL_B), otherwise an old match may be recognized by the scheduler and serviced as a new one.

It is the transition from 0 to 1 in MRL that causes the match actions. Even if other MRL assert conditions are satisfied no action due to a match occurs if MRL was already set to 1, except for MRLE1/2 negation(s). If a match **and** a microoperation negating its corresponding MRL occur during the same microcycle, MRL negation by microcode overrides its assertion. If MRL was already negated before synchronous match and microcode MRL negation any acknowledged captures and pin action occurs anyway.

Regardless of MRL state before, the negation of MRLE(s) flags (the respective one and, in some channel modes, both MRLE1/2) occurs with a synchronous match and MRL negation. Note that MRLE must have been set before (by writing a new match value).

5.8.2.2 Match Enable Flag (MEF)

MEF is a one-bit latch that is unique for all channels in an engine.

MEF can selectively enable assertion of MRL_A/B, depending on the IPAC1/2 field. For IPAC1/2=0xx, MEF=1 enables assertion of MRL_A/B for the scheduled channel during service.For IPAC1/2=1xx, Match1/2 is always enabled, regardless the state of the MEF, but it still depends on the other match recognition conditions. Matches of channels not being serviced are never disabled by MEF.

MEF is not accessible by the microengine or host. MEF is negated at the beginning of time slot transition period for the channel being serviced. After two microcycles into the TST, regardless of TST wait-states, the ME bit in the entry point is copied to MEF to allow selective enabling of MRL for each thread, refer to Section 5.1.2, “Time Slot Transition.” MEF is asserted when no channel is being serviced.

If a channel service needs to postpone a programmed match, MEF assures that microcode wins the race against match event after time slot transition (only for IPAC=0xx).

Note that a match event may be lost during the periods when MEF is negated only if:

- the match comparator is configured for “equal-only” behavior, and
- IPAC1/2=0xx, and
- TCR increments at the rate of system clock divided by 2.

When the comparator is configured as “greater-equal”, the match event that occurred when MEF was negated may be recognized after MEF is asserted again, due to the “greater than” condition.

NOTE

In angle mode the TCR is never clocked faster than system clock divided by 8, so equal-only angle matches are not lost during MEF assertion. See Section 5.10, “eTPU Angle Counter (EAC),” for more details on angle mode.

5.8.2.3 Match Recognition Latch Enable (MRLE1/2)

MRLE1/2 is negated upon the assertion of its respective MRL_A/B. In blocking match channel modes it may also be negated together with the assertion of the twin MRL_B/A. The MRLE1/2 bits ensure that data captured due to the first match event will not be overwritten when MRL_A/B is negated: due to greater-equal comparison, the match condition continues to be true, but should not cause another capture event.

In addition to negation by local match event, the microcode can negate any combination of MRLE1 and MRLE2 to block pending matches. This action will prevent future match events from the selected channel.

Writing the Match1/2 registers by microcode to schedule the next match values sets MRLE1 and/or MRLE2 and enables new matches. This setting overrides the MRLE negation conditions due to channel logic or microcode, see Section 5.8.4, "Channel Modes." By combining write to Match1/2 with MRLE1/2 negation microinstructions, the microcode can negate one MRLE while asserting the other.

The eTPU behaves exactly as the TPU when three conditions are satisfied: the match register is updated (with MRLE already asserted before), the microcode fields MRL_A/B are set to 1 (do not clear), and the MRL_A/B flags are zero. When the eTPU behaves as a TPU a match that comes concurrently with the rewrite of the match register, matching the old value, sets the MRL, as if the setting of the MRLE due to match register write had precedence over its clear by the match at that moment. After this simultaneous operation, the MRLE value stays at 1, and the captured time base value, if any, reflects the match value.

5.8.3 Transition Detection and Time Base Capture

Time base Capture(s) occur when the value of a specified TCR is sampled into the Capture1 and/or Capture2 register. TBS1[1] and TBS2[1] select which TCR will be captured in Capture 1 and Capture2, respectively.

A capture event may occur due to either of the following events:

- the assertion condition of match recognition latch (MRL), even if MRL is simultaneously negated by microcode
- the assertion condition of transition detection latch (TDL), even if TDL is simultaneously negated by microcode.

A capture event occurs together with the assertion of MRL or TDL on T2 positive edge, see Note: . MRL_A/B and TDL_A may, depending on the channel mode, inhibit the capture of the second event's TCR into Capture1/2. As a general rule, values captured by signal transitions are not overwritten by values captured by match events because the match register can always be read, while the capture register value would be lost if overwritten.

NOTE

The assertion of MRL or TDL on the T2 positive edge ensures the capture of the correct TCR1/2 value. In the TPU3, when TCR1 was counting at maximum rate of system clock divided by 2, the next value was captured. See Appendix A, "Microcycle Timing."

The capturing scheme is defined by the channel mode programmed at register PDCM. For more information on mode-dependent capture schemes refer to Section 5.8.4, “Channel Modes.”

5.8.3.1 Transition Detect Latches (TDL_A/B)

TDL_A/B indicate detection of specific transition occurrences on a channel input signal. TDL_A and TDL_B assertion causes service request in single and double transition modes, respectively. TDL_B assertion does not cause service request in single transition modes, and TDL_A assertion does not cause Service request in double transition modes. In single transition channel modes TDL_B can be asserted on the second transition, but does not generate a service request. TDL_B assertion is enabled only if TDL_A is asserted to detect an ordered input signal double transition. The IPAC1 and IPAC2 registers indicate the programmed edges of the first and second detected transition, respectively.

The sampling of a determined value (0 or 1) on the input signal due the occurrence of a match is also treated as a “transition”, depending on IPAC1/2 programming, see Section 5.8.1.2.1, “Input and Output Pin Action Control Registers (IPAC1), (IPAC2), (OPAC1), and (OPAC2).” When using a channel mode where the transition1 is initially blocked and IPAC1 is programmed to detect such “transitions”, the occurrence of a Match1 only unblocks the transition after the sampling. That means IPAC1 configurations 100 and 101 are not effective on modes where transition 1 is enabled by Match1: m2_st, m2_dt, m2_o_st and m2_o_dt, see Section 5.8.4, “Channel Modes.”

TDL_A/B assertion conditions initiates a capture event of one or both selected TCR buses. TDL_A or TDL_B transition event generates a service request, depending on channel mode, previous events and the state of SRI. For more information in service request scheme refer to Section 5.1.1.2, “Entry Point Address Generation,” and Section 5.8.4, “Channel Modes.”

Assertion of TDL_A/B occurs on T2 positive edge. The capture event follows on the same T2, and captures the time base value present when TDL_A/B was asserted, see Note: . The **only** methods of negating TDL_A and TDL_B are during reset and by microcode.

It is the transition from 0 to 1 in TDL that causes the Transition actions: even if TDL assert conditions are satisfied, no action due to a Transition occurs if TDL was already set to 1. However, if a Transition and a microoperation negating TDLs occur at the same time and TDL was already negated, TDL negation by microcode overrides its assertion, but any dependable captures and pin actions occur anyway. This is so an updated capture register can be detected even though the TDL was cleared, and the transition detection is armed for another close following transition.

5.8.4 Channel Modes

The Enhanced Channels support various modes of operation combining Match1/2 recognition and transition detection events which set MDL1/2 and TDL_A/B. The channel mode is individually set for each channel by eTPU microcode, through the PDCM register, see Section 5.8.1.3.2, “Pre-Defined Channel Mode (PDCM).”

The order in which events occur, combined with assigned channel mode, establish which following event detections are inhibited and/or enabled, as well as the actions taken: time base capture, flag setting (MRL_A/B, TDL_A/B), match disabling (MRLE1/2), output signal transition, and service request.

A generic description of channel modes from the usage point of view can be found in Section 5.8.4.1, “Channel Modes Overview.” Each mode is named with a mnemonic acronym for terse reference.

The modes are used differently for input and output signals, as explained in Section 5.8.4.9, “Channel Modes on Input Signal Processing,” and Section 5.8.4.23, “Channel Modes on Output Signal Generation.” Modes also allow combining input processing and output generation in a single channel, as exemplified in Section 5.8.4.31, “Combining Input and Output Signals.”

A dynamic, event-oriented view of each channel mode can be found in Appendix C, “Channel Mode Summary.”

5.8.4.1 Channel Modes Overview

Channel modes are divided on the way they treat transitions in two basic modes:

- Single transition modes (mnemonic suffix _st): in these modes the first transition (flagged in TDL_A) issues a service request, and captures both time bases (selected by TBS1[1] and TBS2[1]) except on sm_st_e.The second transition (flagged in TDL_B) doesn’t issue a service request, but it captures time base selected by TBS2[1], except on sm_st_e.
- Double transition modes (mnemonic suffix _dt): in these modes the second transition (flagged in TDL_B) issues a service request, and each transition captures its own selected time base (Transition 1 and Transition 2 capture time bases selected by TBS1[1] and TBS2[1], respectively).

Transition 2 is always (but not only) enabled by Transition 1, so that transitions are always ordered: TDL_A is set on the first transition and TDL_B on the second. Matches are generally not ordered, except on specific ordered match modes m2_o_st and m2_o_dt. Match capture(s) never override(s) a Transition capture. Transition captures always have precedence over a match capture.

Modes differ mostly by the way matches affect and are affected by other matches and transitions, as explained in next sections. However, some general rules on match blocking apply:

- Blocking of one match by the other is done through MRLEs.
- Matches always block themselves by resetting their own MRLEs (Match1 always blocks Match1, Match2 always blocks Match2).
- Match2 is blocked by first transition (TDL_A) in single transition modes, and by second transition (TDL_B) in double transition modes.
- Both matches are blocked by first transition in single transition modes.

NOTE

The rules above and in following sections may be overruled by the state of the channel latches if they are set/reset by microcode or if channel mode is changed. Care must be taken to change channel modes, and it is advisable to reset channel flags MRL_A/B, TDL_A/B and MRLE1/2 before writing PDCM.

5.8.4.2 Either Match, Blocking Modes (em_b_st, em_b_dt)

In these modes the first match recognition that occurs blocks the other match recognition and generates a service request. They end up with one service request for two programmed match recognitions where only the first match recognition has an actual effect. If both match recognitions occur at the same time, both MRL_A and MRL_B are set, before the mutual blocking takes effect.

5.8.4.3 Either Match, Non Blocking Modes (em_nb_st, em_nb_dt)

In these modes both match recognitions are independent and each match generates a service request. Each match recognition captures its related time base and does not block the other.

5.8.4.4 Match2 Request Modes (m2_st, m2_dt)

In these modes transitions are initially blocked, and are enabled by Match1. Match2 recognition generates the match service request and disables Match1 recognition. Each match recognition captures its own programmed timebase. In case of simultaneous match recognition, both MRL_A and MRL_B are set, and OPAC2 register has priority over OPAC1 for selecting the pin action.

5.8.4.5 Both Match Request Modes (bm_st, bm_dt)

In these modes, a match service request is generated only after both match recognitions occurred. By definition this is a non-blocking match mode: match recognitions do not block each other, implementing a last-served scheme. Unlike other double transition modes, bm_dt blocks Match1 with Transition 2 (not with Transition 1), so that the second transition blocks both matches.

5.8.4.6 Ordered Modes with Match2 Request (m2_o_st, m2_o_dt)

These are ordered match modes on which Match1 recognition must precede Match2 recognition (ordered 1 -> 2). Match1 asserts MRL_A and enables Match2 and transitions. Match2 asserts MRL_B, generates a match service request, and blocks both transitions.

5.8.4.7 Single Match Modes (sm_st, sm_dt)

Single match modes support single or double transition with single match recognition. MRL_B is never set, and MRLE2 has no effect.

5.8.4.8 Single Match Enhanced Mode (sm_st_e)

This is an enhanced single transition and single capture mode, which provides non-filtered input captures in addition to the single capture, allowing one to measure the delay of the digital filter. In an output channel, it has the same functionality of sm_st (captures both time bases at once due to a match recognition).

5.8.4.9 Channel Modes on Input Signal Processing

When processing an input signal, the channel modes can be classified in the following primary mode groups:

- Single Transition, single match: em_b_st, sm_st, sm_st_e
- Single transition, double match: em_nb_st, bm_st, m2_st, m2_o_st
- Double transition, single match: em_b_dt, sm_dt
- Double transition, double match: em_nb_dt, bm_dt, m2_dt, m2_o_dt

In single transition modes, TDL_A assertion may capture both time bases at once, while in double transition modes each transition captures its related time base in its related capture register.

Double transition is always ordered, i.e., TDL_B is enabled by TDL_A and generates the service request.

The channel logic supports various input modes with combinations of single/double transition and single/double match, explained in the following subsections.

5.8.4.10 Either Match, Blocking, Single Transition (em_b_st)

On an input signal, this mode provides double time-out mechanism on a programmed transition edge with two timebases. The signal transition blocks both pending matches, indicating that no time-out condition occurred. The first match recognition blocks the other. This allows the entry table to discern which match recognition caused the first time-out condition, and generates only one service request. Either match performs a double timebase capture. A subsequent transition will overwrite the captures, but the MRL will indicate that the timeout occurred first.

5.8.4.11 Either Match, Blocking, Double Transition (em_b_dt)

In double transition mode each transition is related to one match recognition. TDL_A assertion captures its related timebase, blocks Match1 and enables TDL_B. TDL_B assertion blocks Match2, captures its related timebase and generates a service request. Match recognitions block each other, so if there is a match time-out condition on TDL_A, only one match service request is generated. This mode is good for qualifying two signal transitions by match time-out mechanisms, with one service request. Note that although a TDL_A assertion does not block Match2 recognition, the value captured in Capture1 by TDL_A assertion is not overwritten if a Match2 recognition occurs. The second transition blocks match2. Either match performs timebase captures which don’t overwrite captures by transitions. Thus if a match service request is detected, there has been a timeout. If, in that event, there is also a transition detected, the capture register contains the transition timebases rather than the match timebases.

5.8.4.12 Either Match, Non Blocking, Single Transition (em_nb_st)

On an input signal, this is a double time-out mechanism of independent match recognitions of two different timebases. The match recognitions do not block each other, such that the microcode can check if one or two match recognitions occurred before their related signal transition. The signal transition detection (by IPAC1) asserts TDL_A, blocks both matches, captures both time bases and generates a transition service request, indicating that none of the two time-out conditions occurred. Any combination can be easily resolved by microcode (for example, signal transition after Match1 and before Match2, or signal transition after both Match1 and Match2).

5.8.4.13 Either Match, Non Blocking, Double Transition (em_nb_dt)

Each transition is related to one match recognition in this mode, and the match recognitions are independent of each other. This mode can be used to provide independent time-out conditions for the first and the second signal transition recognitions, and call service in case of any time-out condition.

The first transition detection programmed in IPAC1 sets TDL_A, captures its related timebase, blocks Match1 recognition and enables TDL_B assertion. The second transition detection programmed in IPAC2 sets TDL_B, blocks Match2 recognition, captures its related timebase and generates a service request. Any match recognition that occurs captures its related time base and generates a match service request, independent on the other match recognition.

NOTE

Again, the assertion of a MRL means that the TCR captured due to the match will be in the capture register only if the TDL is not also set.

5.8.4.14 Match2 Request, Single Transition (m2_st)

This mode provides an open window filter for a single signal transition on an input signal. Match1 assertion opens the window, and enables transition detection on TDL_A from this time on. Match2 assertion blocks Match1 (by negating MRLE1), providing conditional window opening. It also generates service request, but does not block transitions, providing a non-blocking time-out mechanism for the estimated signal transition time. If Match2 assertion occurs, this typically indicates a missing transition, or mis-prediction of the transition time.

Transitions can be detected from the microcycle following MRL_A assertion. The transition1 detection asserts TDL_A, blocks Match2, captures both timebases and generates service request.

Using this mode, the channel can replace software open window filtering of qualified transitions by the channel hardware window. The window opening and time-out can be scheduled for any of the two time bases or combination of them. Typically, Match1 will be used to open a prediction window, and Match2 will be used as a time-out condition which does not close the prediction window. This configuration improves noise immunity from early signal transitions, and reduces the probability for blocking late signal transitions due to time-out mis-prediction.

Using these channel conditions, the microcode can easily resolve the state:

- If TDL_A and MRL_A are asserted and MRL_B negated, then the signal transition is in the expected range.
- If MRL_A and MRL_B are both asserted, and TDL_A is asserted, then the signal transition had a time-out condition due to Match2 mis-prediction.
- If MRL_B is asserted and TDL_A negated, a time-out condition occurred, and the expected signal transition had not occurred yet.
- If MRL_A is negated and MRL_B is asserted, then the conditional window did not open at all (for example: a time window is open only after a specific angle, otherwise it is not opened).

5.8.4.15 Match2 Request, Double Transition (m2_dt)

This mode is used as an open window filter for two signal transitions. In this case the Match1 recognition opens the window (unless Match2 recognition occurred first), and Match2 recognition blocks Match1 and generates a match service request. m2_dt is similar to m2_st, but in this case, the second transition blocks Match2. MRL_B assertion is a global time-out condition for the two pulses. Like m2_st, MRL_B can conditionally eliminate the window from opening.

Using the TDL_A, TDL_B, MRL_A and MRL_B conditions, the microcode can easily resolve the state, in a similar manner as m2_st, with additional information on the second transition (TDL_B).

5.8.4.16 Both Match Request, Single Transition (bm_st)

This is a double time-out mechanism for an input signal on two different time bases. Both match recognitions must occur before the signal transition to generate a match time-out service request. Assertion of TDL_A blocks both Match1 and Match2 recognitions, and captures both time bases, indicating there was no double time-out condition from both time bases.

Using the same timebase implements two time-out conditions, the first only sets its related MRL and the second generates a service request. Using the MRL flags allows the microcode to check if one or both match recognitions precede the signal transition.

5.8.4.17 Both Match Request, Double Transition (bm_dt)

In this mode the first transition detection does not block matches and both match recognitions are required to generate a match service request. The second transition detection asserts TDL_B, blocks Match1 and Match2, captures its related timebase and generates transition service request. In this mode, a Match1 recognition which occurs after the assertion of TDL_A does not capture a new value in Capture1, to preserve the actual signal transition time. Assertion of TDL_A, however, always captures its related timebase.

This mode allows putting a double match time-out condition on the second transition.

5.8.4.18 Ordered Mode with Match2 Request, Single Transition (m2_o_st)

This mode provides a closing window filter for a single signal transition on an input channel. Match1 assertion captures its programmed time base in Capture1, opens the filter window (enables assertion of TDL_A), and enables assertion of MRL_B. Match2 recognition captures its related timebase, closes the window (disables assertion of TDL_A) and generates a service request. Due to Match1 and Match2 ordering, the window is opened for at least one microcycle. Match2 recognition indicates a window time-out condition which blocks late signal transitions, outside the prediction window. Transition detection blocks both matches, indicating the transition occurred inside the estimated window. Transitions can be detected from the microcycle following MRL_A assertion until the microcycle on which MRL_B is asserted. When TDL_A is asserted inside the window range it disables both matches, captures both time bases and generates a transition service request.

Using this mode, the channel can replace software window filtering of qualified transitions by the channel hardware window. The window opening and closing time can be scheduled for any of the two time bases or a combination of them.

5.8.4.19 Ordered Mode with Match2 Request, Double Transition (m2_o_dt)

In this mode the channel logic implements a window filter for two detected signal transitions. MRL_A assertion captures its related timebase and enables assertion of both TDL_A and TDL_B. MRL_B assertion captures its related timebase and disables assertion of both TDL_A and TDL_B. Transitions can be detected from the microcycle following MRL_A assertion until the microcycle on which MRL_B is asserted. The first signal transition (following MRL_A assertion) asserts TDL_A, captures its related timebase and enables assertion of TDL_B. The second signal transition detection asserts TDL_B, blocks Match2, captures its related timebase and generates the service request.

If both signal transitions occur inside the scheduled window, Match2 recognition is blocked. If one or both signal transitions do not occur inside the scheduled window, Match2 recognition generates a match service request and blocks further transition detections. The microcode can resolve the state using MRL_A, MRL_B, TDL_A and TDL_B, which affect the microcode entry point selection.

5.8.4.20 Single Match Enhanced Mode (sm_st_e)

This is an enhanced single transition and single match channel mode which provides timing information of the digital filter delay.

The Capture1 register captures the timebase selected by TBS1 due to transition detection specified by IPAC1 or match recognition, as in sm_st mode. Initially, the Capture2 register continuously captures the unfiltered IPAC2-selected signal transitions from the digital filter input, directly from the signal synchronizer. When an IPAC1-qualified, filtered transition detection occurs, TDL_A is set, MRL_A assertion is blocked, and, in addition, captures into Capture2 are also blocked. On service, Capture1 and Capture2 (copied into ERT_A and ERT_B) holds the time of the qualified transition detection (ERT_A), and the time of the last signal transition at the input of the digital filter (ERT_B). Subtracting the time in ERT_B from the time in ERT_A provides the delay of the digital filter.

NOTE

In Channel 0, if ETPUTBCR bit AM=1 (angle mode), the unfiltered input comes from TCRCLK input and the filtered input comes from the TCRCLK filter output. The edge is selected by IPAC1/2, and is independent of the edge selection by ETPUTBCR field TCR2CTL.

5.8.4.21 Single Match, Single Transition (sm_st)

In this mode the channel logic is functionally back-compatible to a TPU3 single action channel, but a match or transition detection captures at once both timebases. The mode recognizes a single transition with single match time-out. Either TDL_A or MRL_A generates service request and captures both timebases. Assertion of TDL_A blocks future assertions of MRL_A.

5.8.4.22 Single Match, Double Transition (sm_dt)

In sm_dt mode, the first transition detection asserts TDL_A, captures a timebase in Capture1 and enables TDL_B. The second signal transition asserts TDL_B, blocks Match1, captures a timebase in Capture2 and generates a service request.

Match1 (before TDL_B) captures into Capture2 the timebase selected by IPAC1, in order not to overwrite the captured value of TDL_A.

This mode is used for scheduling one time-out condition on two input signal transitions (pulse time-out).

5.8.4.23 Channel Modes on Output Signal Generation

Since channel logic can generate output signal transitions based on matches, the channel can be viewed as working in the following primary mode groups for signal generation:

- Single match: em_b_st, sm_st, sm_st_e, em_b_dt, sm_dt
- Double match: em_nb_st, bm_st, m2_st, m2_o_st, em_nb_dt, bm_dt, m2_dt, m2_o_dt

The channel logic supports various match channel modes with single/double match, as explained in the following subsections.

5.8.4.24 Either Match, Blocking Modes (em_b_st, em_b_dt)

On an output signal these modes are useful when using two different time bases to set a required signal transition. The first match condition which is met sets a required pin action, captures both time bases, blocks any effects of the other recognition, and generates a service request. Because the first match recognition blocks the other, the microcode can get good separation in the function entry table as to which match caused the time-out first, and both time bases are captured, enabling the microcode to compare one timebase to the other at the moment of the match recognition. These modes can be used for:

- Scheduling a required pin action to the first match recognition of two different time bases.
- Cancelling a programmed pin action scheduled on one time base by match on another timebase (as a consequence of Table 5-9). The microcode has to set the OPAC register of the cancelling match to no-action and the OPAC register of the other match to the required pin action which may be blocked. If both matches are recognized at the same time, and Match1 is the cancelling match, then the pin action is blocked, since Match1 has priority in this case. If Match2 is the cancelling match, it does not block the pin action in case of two matches at the same time.

5.8.4.25 Either Match, Non Blocking Modes (em_nb_st, em_nb_dt)

On an output signal these modes are useful in combination with the ME bit set on the entry point, to define an interlaced operation. For example, each match recognition can set a pin action, and the second pin action is not sensitive to microcode latency (ME bit asserted). Example for usage is PWM interlaced function on which the latency is determined by the period and not the duty cycle.

Another possibility is using one match for pin actions and the other match for an unrelated timed task without pin action (double the functionality of a single channel).

5.8.4.26 Match2 Request Modes (m2_st, m2_dt)

These modes can generate narrow pulses or do conditional pin actions on an output signal. A conditional pin action means that the pin state is changed only if the match recognitions occurred in the correct order. In these modes Match2 recognition generates the service request, has priority over the pin action, and blocks future Match1 recognitions.

Setting OPAC1 to a desired pin action and OPAC2 to no-action, and using different time bases for Match1 and Match2 defines a conditional OPAC1 pin action which can be blocked by Match2 recognition. For example, setting Match1 on time and Match2 on angle can limit the pin action to a maximum angle value.

When pulses are generated, the service is requested at the trailing edge of the pulse, after MRL_B is asserted.

5.8.4.27 Both Match Request Modes (bm_st, bm_dt)

On an output signal, each match recognition can affect the pin state, and capture its programmed time base. This way the pin action can be programmed separately for both match recognitions. For example, both match recognitions can negate the signal, and service request is generated after both conditions are met. This mechanism can set two conditions to do a required pin action, and the first recognition changes the signal, but service is called only after both conditions occur.

When using the same time base, these modes can generate narrow pulses in any required order.

Another usage is generating a required pin action on one programmed time and service request later on another time, after the second match recognition occurs, or capturing some timebase on one time and generating a required signal transition and service request later.

5.8.4.28 Ordered Modes with Match2 Request (m2_o_st, m2_o_dt)

The order of the match recognitions imply that OPAC1 register programmed pin action always precedes the OPAC2 register pin action. Setting OPAC1 to no-action, based on the greater-equal comparator, enables using Match1 on one time base to delay the signal effect of Match2 on the other time base. This method implements a conditional pulse extension or conditional delay on signal transition.

These modes can also be used for deferred pulse generation with microcode service request after its trailing edge (if Match1 condition comes after Match2 condition). Another option is having Match1 recognition associated with output pin actions and Match2 recognition for a timed microcode task, such as calculating the next pulse in a recurring sequence, which has to be scheduled at a programmed time that may be delayed by the Match1 pin action.

5.8.4.29 Single Match Modes (sm_st, sm_dt, sm_st_e)

There is no difference between plain and enhanced single match modes on an output signal.

This mode’s channel logic is functionally back-compatible with a TPU3 single match output channel. Match1 recognition generates service request and sets the pin state according to OPAC1 register. Match1 instantaneously captures the timebase selected by TBS1 in Capture1 and the timebase selected by TBS2 in Capture2.

5.8.4.30 Match/Transition Pin Action Conflict Resolution

Matches cause pin actions define by OPACs. Pin actions cause captures according to IPACs. Some modes combine them. The combinations are more complicated with separate pins for input and output. For more information, see Section 5.8.1.2, “Pin Control Registers.” Simultaneous matches/transitions may be associated with different, possibly contradictory, pin actions. Table 5-9 illustrates how these conflicts are resolved.

If an OPAC1/2=000 (no action) prevails over the other non-zero value OPAC, as shown in Table 5-9, then a simultaneous Match1/Transition1 and Match2/Transition2 causes no output pin action to occur. Therefore, a match on the action logic with OPAC=000 inhibits simultaneous actions of the other lower priority OPAC, see Table 5-9. This priority scheme also applies when output actions are caused by inputs (OPAC=1xx).

Table 5-9. Simultaneous Match Pin Action Priority

Channel Submode	Priority
em_nb_st / em_nb_dt	OPAC1
em_b_st / em_b_dt	OPAC1
bm_st / bm_dt	OPAC1
m2_st / m2_dt	OPAC2
m2_o_st / m2_o_dt	in these modes there is no possibility of simultaneous matches

5.8.4.31 Combining Input and Output Signals

The processing of input signal can be combined with output signal generation. A detected input transition can trigger an output signal edge, without microcode intervention, by using OPAC options 1xx.

The channel set-up examples below show these two capabilities combined (see Figure 5-5).

The first example implements a fast (no microcode intervention) short-circuit protection feedback mechanism for driving current output devices. The signal, which is driven by the channel output, after the current driver feeds back to the channel input. The input signal is normally delayed from the output signal by the device turn-on delay. After the channel output turns on, the channel logic must check if the driver output (connected to the channel input) follows the driven value after the maximum device turn-on delay. If it doesn’t, the channel has failed, and the channel output must be turned off immediately to avoid damaging the device.

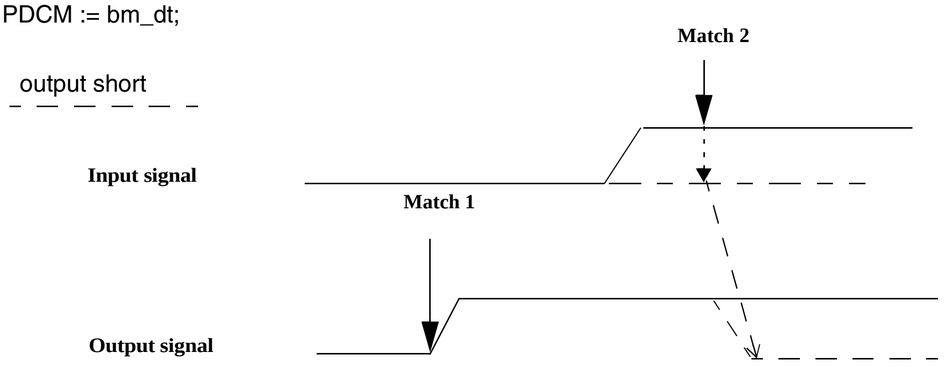
Note that, because IPAC1=001, the actual input transition time gets captured into the Capture1 register when the output is not shorted, so one can measure the actual driver turn-on delay by subtracting Match1 from Capture1.

NOTE

When IPAC = 1xx, a match event can cause a simultaneous match recognition and a transition detection. Depending on the channel mode, a match and transition may have conflicting effects on other transition/match blocking or enabling. In these cases, blocking always prevails over enabling, effective on the next microcycle.

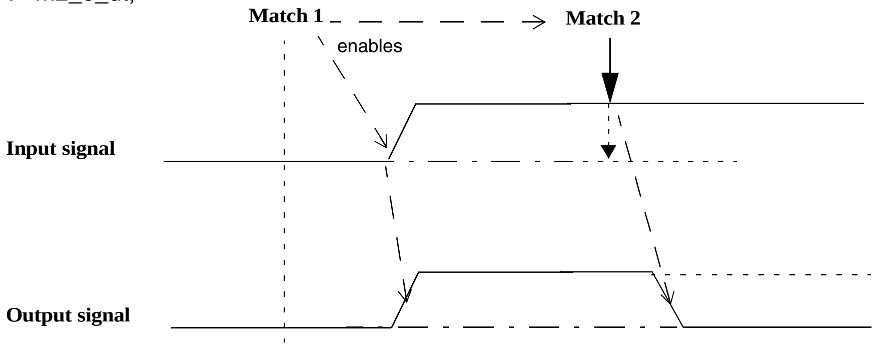
Example 1: Short-circuit protection feedback

IPAC1 := 100; OPAC1 := 001; Match1 := output activate time;
IPAC2 := 100; OPAC2 := 100; Match2 := Match1 + max. high-current driver turn-on delay;
PDCM := bm_dt;



Example 2: Pulse generation on windowed input transition

IPAC1 := 001; OPAC1 := 101; Match1 := window open time;
IPAC2 := 101; OPAC2 := 100; Match2 := window close time, input sampling;
PDCM := m2_o_dt;



Example 3: Pulse generation on input sampling

IPAC1 := 100; OPAC1 := 100; Match1 := window open time,input sampling;
IPAC2 := 000; OPAC2 := 001; Match2 := window close time = Match1 + pulse width;
PDCM := em_nb_dt;

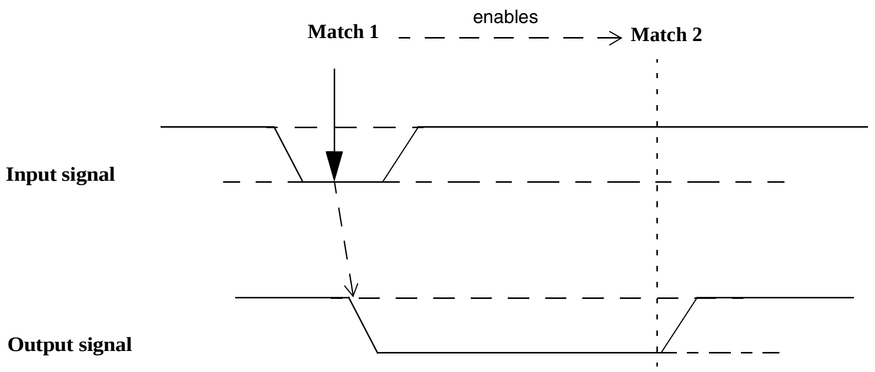


Figure 5-5. Input/Output combination

5.8.5 Channel Link

A channel can issue service requests to other channels through microcode, by assigning to the write-only microengine register LINK, refer to Section 8.2.6, “LINK Register,” a value which specifies the target channel of the link service request, as shown in Figure 5-6.

Writing to the LINK register issues a link request to the target channel, setting its LSR flag. Each channel has its own LSR flag, which can be tested as a microcode branch condition and reset through the microcode field LSR. The link branch condition is sampled at the TST start, with the value used to calculate the entry point.

Writing LINK with another channel target value in the same thread issues a link service request to the new target, without negating the service request to the former one. This allows a channel to issue service requests to any number of channels, including itself. Neither LINK nor LSR microengine accesses are qualified by the CHAN register, they always access the serviced channel LINK and LSR regardless of the value written in CHAN.

If the microcode executes an instruction with field LSR=0 (clear link service request), the link branch condition is cleared. However, the link service request itself is cleared only if no link was received by the serviced channel during the same thread, see Note: . If the microengine clears the LSR of its channel and, simultaneously, link service request is issued to the current serviced channel, the branch condition is cleared but the link service request remains pending.

NOTE

This can only happen if the link service request came from the other engine or from the serviced channel itself.

7	6	5	4	3	2	1	0
Engine Selection		Channel Number					

Figure 5-6. Microengine LINK Register

A channel can issue link service requests to channels in any of two engines, determined by the LINK register field engine selection (2 bits), as shown in Table 5-10.

Table 5-10. LINK Engine Selection

Engine Selection	Description
00	Select This Engine
01	Select Engine A
10	Select Engine B
11	Select Other Engine

The engine which receives the link cannot distinguish where the link comes from, except by some user-programmed protocol using SPRAM parameters.

5.8.6 Enhanced Digital Filter (EDF)

The EDF eliminates passing of signal transitions which are caused by noise. Its purpose is to eliminate false transition service requests caused by noise pulses which are shorter than a programmed width.

The EDF has three modes of operations, selected by the CDFC field in the ETPUECR register, see Section 4.2.4, “eTPU Engine Configuration Register (ETPUECR).” These modes offer selections of trade-off between noise immunity and signal latency. Table 5-11 gives an example of minimum detected signal pulse and maximum filtered noise pulse in the three EDF operation modes. In angle mode the EDF in channel 0 is replaced by the digital filter and synchronizer of the TCRCLK signal. In this mode, channel 0 works in combination with the Angle Counter logic, and their operation is fully synchronized.

Following subsections provide the functional description of the eTPU channel digital filter.

5.8.6.1 Two-Sample Mode

In this mode the EDF works like the TPU2/3 digital filter. It uses the filter clock which is the system clock divided by (2, 4, 8,..., 256) as a sampling clock. The filter clock is selected by the FPSCK field in the ETPUECR engine configuration register, see Section 4.2.4, “eTPU Engine Configuration Register (ETPUECR).” The EDF compares two consecutive samples. If both samples have the same value, the input signal state is updated. Note that the EDF works like the TPU1 four-(system)-clock digital filter if the FPSCK field selects the system clock divided by two.

5.8.6.2 Three-Sample Mode

In this mode, like in the TPU2/3 mode, the EDF uses the filter clock as a sampling clock. The EDF compares three consecutive samples. If all three samples have the same value, the input signal state is updated.

The three-sample mode has more signal latency than the two-sample mode, but also better noise immunity and better ratio between minimum detected signal pulse to maximum filtered noise pulse.

5.8.6.3 Continuous Mode

In this mode the EDF compares all the values sampled at the rate of the system clock divided by two, between two consecutive filter clock pulses. If the signal is continuously stable for the entire digital filter clock period (i.e all the samples have the same signal value), the input signal state is updated.

This method has the same latency and the same ratio between minimum detected signal pulse to maximum filtered noise pulse, as the two-sample mode, as long as there is no noise. Each sampled noise delays the signal transition detection by at least a whole digital filter clock period.

The continuous mode gives the best noise immunity by comparing multiple samples of the noise. On the other hand, when a short noise pulse appears in the middle of the filter clock period at the same time of a real signal transition, the continuous mode may reject a real signal transition and delay the response to the first filter clock period in which the signal is continuously stable. This may add to the latency and increase the minimum required duration to detected a signal pulse in a noisy environment.

5.8.6.4 Filter Clock Prescaler

The TCRCLK signal and each channel configured as an input have an associated synchronizer followed by a digital filter connected to the signal that samples signal transitions. After reset, the digital filter filters out high and low pulse widths smaller than the period of two system clocks, preventing these transitions from being input to the transition detect logic. The synchronizer and digital filter are guaranteed to pass pulses that are greater than or equal to the period of four system clocks. By changing the FPSCK field in register ETPUECR the user can select a lower clock rate for the filter signal to define wider valid pulses and filter out wider noise pulses. The filter prescaler clock control is a division of the system clock. To guarantee pulse detection by the digital filter, the pulse must cover at least the stated number of samples at the filter clock rate. For example, a two sample digital filter must sample two points in the pulse to detect it.

Table 5-11. Minimum Detected Pulse / Maximum filtered Pulse Width

Filter Control	Sample on System Clock Divided by:	Min. Width Detected / Max. Width Filtered[1,2]	
		Two Samples or Continuous Mode	Three Samples Mode
000	2	4 / 2	6 / 4
001	4	8 / 4	12 / 8
010	8	16 / 8	24 / 16

Table 5-11. Minimum Detected Pulse / Maximum filtered Pulse Width

Filter Control	Sample on System Clock Divided by:	Min. Width Detected / Max. Width Filtered[1,2]	
		Two Samples or Continuous Mode	Three Samples Mode
011	16	32 / 16	48 / 32
100	32	64 / 32	96 / 64
101	64	128 / 64	192 / 128
110	128	256 / 128	384 / 256
111	256	512 / 256	768 / 512

1 This table shows pulse widths in number of periods of the system clock.

2 Minimum Width *Guaranteed* to be Detected / Maximum Width *Guaranteed* to be Filtered. For instance, in two-sample mode: a pulse of 3 clocks may be detected or filtered, depending on its phase against the filter clock.

5.9 Time Bases

Each eTPU engine has two time counter registers, TCR1 and TCR2. They provide 24-bit time bases, shared by all 32 channels. Any channel can use both time bases to:

- Match channel's internal registers Match1 or Match2;
- Capture time base value to channel's internal registers, Capture1 and/or Capture2, when a match recognition or an Input transition detection occurs. For more information on channel events refer to Section 5.8, "Enhanced Channels."

The TCR1 and TCR2 counters are accessible by the microcode for read and write operations. TCR1 and TCR2 values are updated in T2 and read in T4, see Appendix A, "Microcycle Timing." Both TCR1 and TCR2 values can be imported from or exported from one eTPU engine to another one only, see Note:

NOTE

The TCR1/2 counters between the two engines are out of phase by 1 system clock, even when time bases are shared between them. This is due to the interleaving of the eTPU engines. Even though the timers are out of phase by a system clock, all channels are in phase with respect to eTPU microcycles.

5.9.1 Timer Count Register 1 (TCR1)

TCR1 can be used in the following modes:

- Internally clocked mode
- Externally clocked mode

The host program can read TCR1 time base through the ETPUTB1R, see Section 4.3.2, "eTPU Time Base 1 (TCR1) Visibility Register (ETPUTB1R)."

The TCR1 bus runs through all the local engine channels. In channels which select TCR1 as Match1 and/or Match2 source, when its value is greater or equal to the programmed match value, a match1 and/or match2 event occurs on that channel. A recognized match event sets its related match recognition latch 1 or 2, and according to the pre-defined channel modes (PDCM) it may generate a channel service request. For details on eTPU channels refer to Section 5.8, “Enhanced Channels.”

When a match or signal transition event is detected by any of the channels, it performs a capture operation. Any channel can capture the TCR1 value in its Capture1 and/or Capture2 register due to the event. This way the microcode gets a time tag of the event without channel service latency.

5.9.1.1 Externally Clocked Mode

TCR1 can be driven externally by the TCRCLK input, after the digital filter. The TCR1 clock source is configured by the TCR1CTL bit, as shown in Figure 5-7. For more information on clock source selection, please refer to Section 4.3.1, “eTPU Time Base Configuration Register (ETPUTBCR).”

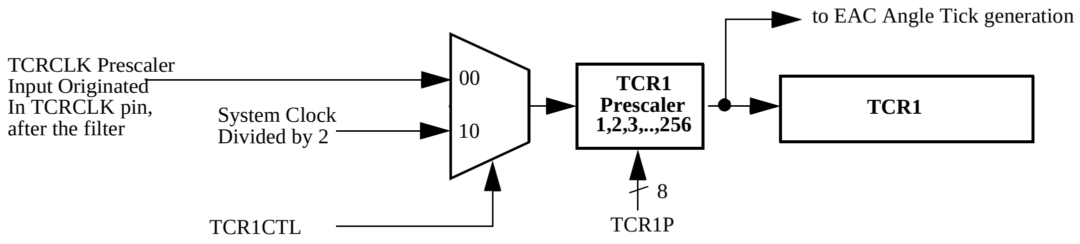

to EAC Angle Tick generation
TCRCLK Prescaler Input Originated In TCRCLK pin, after the filter
00
System Clock Divided by 2
10
TCR1CTL
TCR1 Prescaler 1,2,3,..,256
8
TCR1P
TCR1

Figure 5-7. TCR1 Clock Selection

5.9.1.2 Internally Clocked Mode

TCR1 can be driven by the system clock divided by 2. TCR1 can also be clocked by a peripheral timebase clock generated within the MCU and is selected by TCR1CTL. The clock source selected by TCR1CTL is prescaled by a factor of 1 to 256, selected by ETPUTBCR field TCR1P. For more information on prescaler configuration refer to Section 4.3.1, “eTPU Time Base Configuration Register (ETPUTBCR)**.”**

5.9.2 Timer Count Register 2 (TCR2)

The TCR2 is a 24-bit counter which can be used in the following modes:

- Pin transition mode: Count the rise, fall or both transitions of TCRCLK signal.
- Angle clock mode: In the Angle Clock mode, TCR2 counts and times the signals from a toothed wheel and with software support, tracks the angle of the wheel. This

implements an Angle PLL, and generates angle information to the channels. This mode is targeted for angle based applications, such as internal combustion engine control.

- Gated mode: Count with rate derived from the system clock divided by eight. The TCRCLK signal is used to gate this count, enabling pulse accumulator operations.
- Internally clocked modes: TCR2 is driven by the internal clock, with count rate of the system clock divided by eight.

All clock sources pass through a prescaler. The TCR2 timebase may also originate from the EAC which is a hardware angle counter. The EAC is activated in angle mode, and can generate one of two angle clock formats on the TCR2 bus. Figure 5-8 shows the diagram for TCR2 clock control.

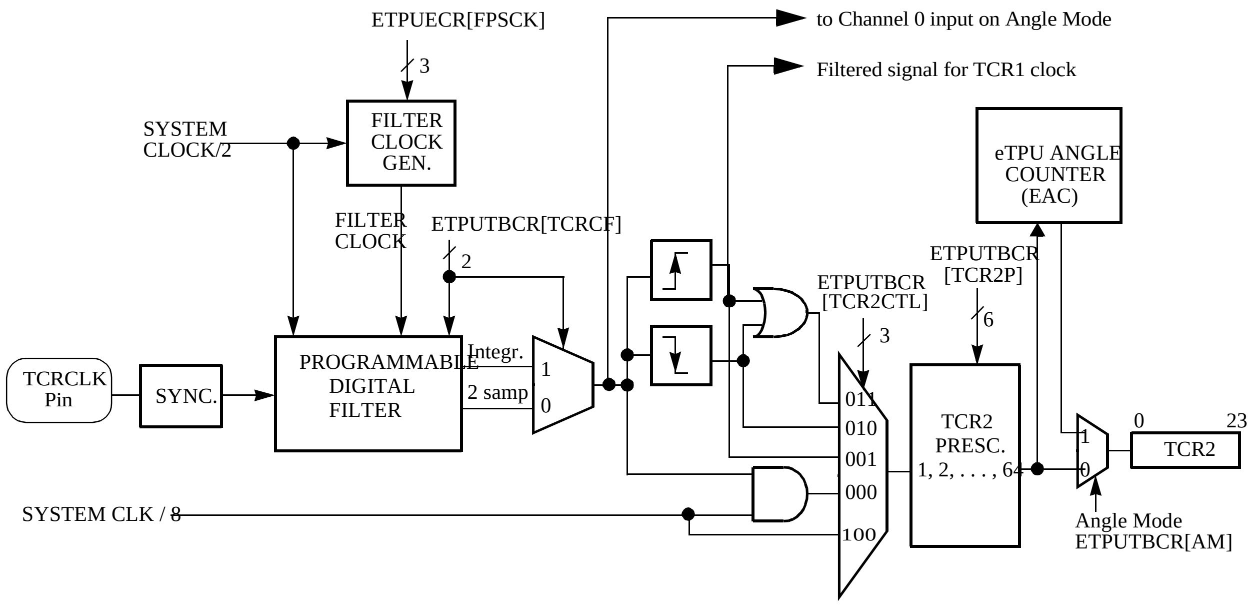


Figure 5-8. TCR2 Clock Control

The TCRCLK signal input is passed through a synchronizer and a programmable digital filter. In angle mode, the synchronizer and filter are also used in channel 0, replacing channel 0's input synchronizer and filter, to get the same timing in the EAC and Channel 0.

NOTE

The angle counter is a hardware system supported by software and hence requires that the signal be applied to Channel 0.

The TCRCLK synchronizer is a dedicated filter that provides best latency while maintaining proper noise filtering and is configured through the TCRCF field, see Section 4.3.1, "eTPU Time Base Configuration Register (ETPUTBCR)."

The TCR2 bus runs through all the local engine channels. It transitions on clock T2. For more detail, see Appendix A, "Microcycle Timing."

The TCR2 counter is accessible by the microcode for read and write operations. Its current value is used for getting the current counter value (representing signal transitions, time or angle), and the captured values are used for channel relative count calculations. The TCR2 value is readable to the host through the ETPUTB2R register, refer to Section 4.3.3, “eTPU Time Base 2 (TCR2) Visibility Register (ETPUTB2R).” When the TCR2 bus value is imported from the STAC bus, TCR2 is not writable by the microcode, and read access from the ‘microcode or from the host reflect the imported TCR2 value.

5.9.2.1 TCR2 Clock Prescaling

Any clock source selected by TCR2CTL is prescaled by a factor of 1 to 64, selected by ETPUTBCR field TCR2P. For more information on prescaler configuration refer to Section 4.3.1, “eTPU Time Base Configuration Register (ETPUTBCR).”

5.9.2.2 TCR2 Gated Mode

TCR2 Gated mode is selected in field TCR2CTL of register ETPUTBCR. In this mode the TCRCLK signal enables or disables transfer of the system clock divided by 8 to the TCR2 prescaler. By programming the prescaler, TCR2 can run at rates from system clock divided by eight down to system clock divided by 512, in steps of eight system clock divisions. For more information refer to Section 4.3.1, “eTPU Time Base Configuration Register (ETPUTBCR).”

5.9.2.3 TCR2 Signal Transition Modes

These modes are selected when the TCR2CTL field in ETPUTBCR is set to rise, fall or “rise-and-fall”. In these modes the TCRCLK signal is the TCR2 clock source, and its maximum transition rate depends on the TCRCLK digital filter mode of operation. The TCRCLK digital filter can be programmed to use the system clock divided by two, or use the same filter clock of the channels, controlled by the TCRCF field in ETPUTBCR. It contains an up-down counter which operates as a digital integrator, optimizing signal latency in the selected mode and clock rate.

When the system clock divided by two is selected, the synchronizer and the digital filter are guaranteed to pass pulses that are wider than four system clocks (two filter clocks). Otherwise the TCRCLK is filtered with the same filter clock as the channel input signals. For details on TCRCLK and channels digital filter control refer to Section 4.3.1, “eTPU Time Base Configuration Register (ETPUTBCR),” and Section 5.8.6, “Enhanced Digital Filter (EDF).”

5.9.2.4 TCR2 Bus in Angle Clock Mode

In this mode the TCR2 counter operates as part of the eTPU angle counter (EAC). The TCR2 bus value reflects this angle representation in which it counts angle ticks. Angle mode is selected when the AM bit is set in ETPUTBCR.

Note that when TCR2 works in angle mode, it does not count directly from the TCR2 clock input which indicates tooth signal transition. Its angle counter is controlled by the count control and high rate logic, see Section 5.10.5, “Count Control and High Rate Logic,” which provides the interpolated tooth position, and with software support, handles cases of a missing tooth/missing teeth, acceleration, deceleration and mechanical corrections.

The EAC uses the TCRCLK signal to get tooth transition indications. The TCR2CTL field in ETPUTBCR has to be set for the appropriate tooth edge detection rise, fall or “rise-and-fall”. TCR2 count clock comes from the EAC control and not directly from the physical tooth. This way the eTPU is able to processes signal transitions and handle missing teeth and flywheel mechanical corrections.

In angle mode, eTPU channel 0 operation is combined with the EAC operation. The TCRCLK digital filter is used both by the EAC and by channel 0 to get full synchronization between the two logics.

The eTPU angle counter (EAC) logic runs continuously and updates the TCR2 angle counter without latency as long as it is maintained by the angle count software.

5.9.3 Shared Time and Angel Count (STAC) Bus Interface

Both time bases TCR1 and TCR2 can be shared among the engines and with the eMIOS block in the MPC5554. Either of the eTPU engines or the eMIOS can drive their time bases to the STAC interface, acting as a server, while another block can use the bases and behave like a client. For further information on the TCR shared bus operation refer to Section 4.3.4, “STAC Bus Configuration Register (ETPUREDCR).”

The eTPU can export to or import from the STAC bus interface either of the following:

- TCR1: When TCR1 is imported from the STAC bus, it becomes read-only for the microcode and reflects the imported values. For details refer to Section 5.9.1, “Timer Count Register 1 (TCR1).”
- TCR2: When TCR2 is imported from the STAC bus, it becomes read-only for the microcode, and reflects the imported values. When exported to the STAC bus, TCR2 can work in either angle mode or as a free running counter associated with the TCRCLK signal. For details, refer to Section 5.10, “eTPU Angle Counter (EAC).”

The configuration of ETPUTBCR[AM] and ETPUREDCR[REN2, RSC2] selects the IP bus interface driver. Table 5-12 describes these configurations.

Table 5-12. Interface Bus and Host Read Sources

ETPUTBCR[AM]	ETPUREDCR[REN2, RSC2]	TCR2 Bus Source (Host read of ETPUTB2R)	STAC Bus Interface Driver
0	0x (disabled)	TCR2/Time	x
1	0x (disabled)	TCR2/Angle	x
0	11 (Server)	TCR2/Time	TCR2/Time
1	11 (Server)	TCR2/Angle	TCR2/Angle
1	10 (Client)	Forbidden[1]	

[1] STAC client configuration in angle mode is also forbidden for TCR1.

Note that when TCR2 is a STAC bus client, the ETPUTBCR bit AM has no effect for eTPU configuration since EAC operation is disabled; angle mode is not available for STAC bus clients. When TCR2 is a stand-alone counter or a STAC bus server, the same value that is driven to the internal TCR2 bus is also exported to the STAC bus (either Time Count or Angle).

STAC bus interface configuration is provided by the ETPUREDCR bits REN1, REN2, RSC1, and RSC2. REN1 and REN2 enable the time and count IP interface to interact with the resource (either TCR1 or TCR2 bus). RSC1 and RSC2 configure the resource (either TCR1 or TCR2 bus) as server or client.

5.9.4 Global Time Base Enable (GTBE)

GTBE bit in ETPUMCR register enables time bases in both engines, allowing them to be started synchronously. In addition, assertion of the GTBE bit also enables the time base for the eMIOS, thus asserting ETPUMCR[GTBE] synchronously enables eTPU A, eTPU B, and eMIOS time bases.

The eMIOS on the MPC5554 also has a separate and independent GTBE bit. Asserting the eMIOS GTBE bit has the same effect as asserting ETPUMCR[GTBE], i.e. eTPU_A, eTPU_B, and eMIOS time bases are all synchronously enabled. To globally disable the time bases, the GTBE bits in both eTPU and eMIOS must be negated.

5.9.5 TCRCLK Digital Filter

The TCRCLK signal has an improved integrating digital filter with a 2-bit up-down counter. The counter counts up to 3 when a high signal level is detected, or down to 0 when a low level is detected. The signal state is updated to one when the counter stops at 3, or zero when the counter stops at 0. The field TCRCF in register ETPUTBCR, see Section 4.3.1, “eTPU Time Base Configuration Register (ETPUTBCR),” determines whether the TCRCLK signal input (after a synchronizer) is filtered with the same filter clock as the channel input signals, see Section 5.8.6, “Enhanced Digital Filter (EDF),” or uses the system clock divided by 2. The TCRCF field also sets the TCRCLK digital filter to work in integrator mode or the same two sample mode as the channel filters (seeTable 4-10).

5.10 eTPU Angle Counter (EAC)

5.10.1 General

The EAC logic contains logic which allows it to track the angle of a toothed wheel by processing the timing of the teeth, detected by a sensor. This hardware works in combination with the TCRCLK signal, the TCR2 counter and Channel 0 to generate angle information on the TCR2 bus which is passed to all the local engine channels. The EAC helps to implement a digital angle PLL (see Figure 5-12), which combines hardware with microcode processing at channel 0. The angle measurement is based on history knowledge of the tooth period, for predicting the period of the next tooth. The tooth period is partitioned into a programmable number of **Angle Ticks**. The eTPU application will use the divider in the MAC/Divide unit to calculate an integer and a fraction part of the angle tick such that the full tooth period gets the correct programmed number of angle ticks with no accumulated error.

Every tooth can be divided in angle ticks, up to 1024. In a 60 tooth flywheel, 128 Angle Ticks per tooth provide resolution of ~0.05 degrees per tick, which meets the accuracy requirement of 0.1 degrees in current automotive applications.

The measurement of one tooth in angle ticks is independent of engine RPM; it is the tooth period itself (and the corresponding tick period) that is re-calculated for each new tooth, based only on the period of the last tooth span.

For angle measurement applications, eTPU channel 0 is dedicated to service physical tooth detection, sharing the same filter as the TCRCLK signal to get the same timing in the EAC and Channel 0.

The EAC supports deceleration, acceleration, last tooth and missing tooth scenarios. In case of a missing tooth, the EAC can be configured to insert a dummy tooth or to simply measure a longer tooth.

Figure 5-13 shows the block diagram of the Angle Counter system. TCR1 is used as a time base which measures the tooth period and is used for partitioning the period to angle ticks.

5.10.2 Angle Mode Registers

In angle mode, the registers described below control eTPU angle operations. They are accessible only by the microengine as source and destination registers in microinstructions.

When the eTPU is not in angle mode (AM bit is reset in ETPUTBCR register), all angle mode registers can be used as general purpose registers.

5.10.2.1 Tooth Program Register (TPR)

TPR provides configuration for the angle counter circuit. In this register, the microcode can properly adjust the tooth count (controlling last tooth, missing teeth, dummy tooth insertion, halt until tooth detection) and the number of angle ticks per tooth (field TICKS). Note that this register is sampled into a temporary register in the EAC logic when the high rate mode is detected, see Section 5.10.5.3, “High Rate Mode (Acceleration),” which means that changes to this register may take effect only for the next tooth.

Refer to Section 5.10.5, “Count Control and High Rate Logic,” and sections 5.10.6 to 5.10.9 for a detailed explanation about the use of this register. Figure 5-9 provides a detailed description of the TPR register.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	LAST	MISSCNT		IPH	HOLD	TPR10	TICKS									
RESET:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-9. TPR Register

Table 5-13. TPR Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	15	LAST	Last Tooth Indication. Asserted by microcode and negated at the end of the tooth period. 1 Last Tooth: reset TCR2 on the next physical tooth edge (or when IPH=1) when MISSCNT=0. 0 Not Last Tooth.
R/W	14 – 13	MISSCNT[1:0]	Missing Tooth Counter. Decremented on each estimated tooth, stops at zero. Used for generation of “Dummy Tooth” whenever it holds a non-zero value (see Table 5-14).

Table 5-13. TPR Bit Field Descriptions (continued)

Read/Write	Bits	Name	Description
R/W	12	IPH	Insert Physical Tooth. This bit is used for exiting halt mode[1] which is caused by missing a detection of a physical tooth. It generates a dummy physical tooth which has the same effect as a real physical tooth, and resets itself subsequently. If EAC is in halt mode, it switches back to normal mode[2]. If EAC is in normal mode, it switches to high rate mode. 1 Insert dummy physical tooth. 0 No Operation. **Note:** If the EAC is in Normal mode and IPH is asserted, the EAC stays in Normal mode for 1 microcycle more before going high-rate. **Note:** IPH reads as 1 in the next microinstruction after it is asserted, negating subsequently. However, it can be set twice in two consecutive microinstructions to generate two teeth and make the EAC go from Halt to Normal to High Rate Mode.
R/W	11	HOLD	Force EAC Halt. This bit forces the EAC to halt its operation in a special EAC Halt mode until a new physical tooth is detected. Assertion of this bit immediately halts the EAC in the middle of the tooth period. When a new physical tooth is detected, the bit is automatically negated by the EAC. The HOLD bit can be used for synchronizing the EAC tooth count, in case that a false physical tooth is detected due to noise. 1 Force EAC to halt until detection of a physical tooth. 0 Normal Operation.
R/W	10	TPR10	Reserved in Angle Mode, must always be written 0. When Angle Mode is off, can be used as general purpose register bit, like the rest of the register.
R/W	9 – 0	TICKS[9:0]	Angle Ticks Number in the Current Tooth. This field defines the number of angle ticks in the current physical tooth. It partitions the tooth period to the required number of angle ticks. The number of ticks in the tooth span is TICKS, but the number of ticks in a tooth period is TICKS+1. The last count is the Tooth.

[1] EAC Halt mode has nothing to do with microengine Halt state. See Section 5.10.5.2, "Halt Mode (Deceleration)."
[2] the missing of a physical tooth naturally causes EAC to get into Halt mode.

Table 5-14. MISSCNT Values

Value	Meaning
00	Not a Missing Tooth
01	One Missing Tooth
10	Two Missing Teeth
11	Three Missing Teeth

NOTE

MISCNT can only be written a non-zero value if LAST is written 1 simultaneously.

NOTE

In High Rate mode, TPR writes are immediately effective only for bits IPH and HOLD. All other fields are sampled in a shadow register and become effective at next tooth if not in High Rate mode, or when EAC leaves High Rate mode. However, if TPR is written a second time right after IPH is asserted in Normal mode, this second write behaves as if EAC is still in Normal mode. Only in the next microcycle (after execution of a nop, for instance) the TPR writes are shadowed, acknowledging High Rate Mode. The value read by microcode is the same written in any situation.

NOTE

Bits LAST, IPH, and HOLD must not be asserted all at once. MISSCNT can only be rewritten after it finished the countdown to 0.

5.10.2.2 Timer Counter 2 (TCR2)

In angle mode TCR2 counts teeth and angle ticks, instead of time.

	23	22	21	20	19	18	17	16
R	Angle Tick Counter[8:16]							
W								
RESET:	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	Angle Tick Counter[15:0]															
W																
RESET:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-10. TCR2 in Angle Mode

This 24-bit free-running counter is used to generate an accumulated angle fraction value. It is updated by the angle tick generator, refer to Section 5.10.4, “Angle Tick Generator,” for more details. Refer to Section 5.10.5, “Count Control and High Rate Logic,” for a detailed explanation about the use of this register in angle mode.

TCR2 provides continuous count of the angle in units of angle ticks. The angle tick counter in TCR2 can be reset due to “Last Tooth” microcode indication and can be written by microcode at any time.

5.10.2.3 Tick Rate Register (TRR)

The exact period of the angle tick is programmed in the tick rate register by microcode. The period of the angle tick is given in units of TCR1 clocks. Refer to Section 5.10.4.1, "Calculating the Angle Tick Period Integer and Fraction," for a complete description about the mechanism to calculate the value to be written into TRR register.

	23	22	21	20	19	18	17	16
R	INTEGER[14:7]							
W								
RESET:	0	0	0	0	0	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	INTEGER[6:0]							FRACTION								
W																
RESET:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5-11. TRR Register

Table 5-15. TRR Register Bit Field Descriptions

Read/Write	Bits	Name	Description
R/W	23 – 9	INTEGER[14:0]	The integer part of TCR1 clocks in one Angle Tick. This number, decremented by one, is a down-counter preload value. A value of INTEGER=0 represents an integer of 32768.
R/W	8 – 0	FRACTION[8:0]	Nine-bit fractional part of TCR1 clocks in one angle tick. The FRACTION value is accumulated in the EAC Fraction Accumulator, and whenever the result overflows (i.e., the accumulated fraction added up to an integer), the tick prescaler is halted for one TCR1 clock.

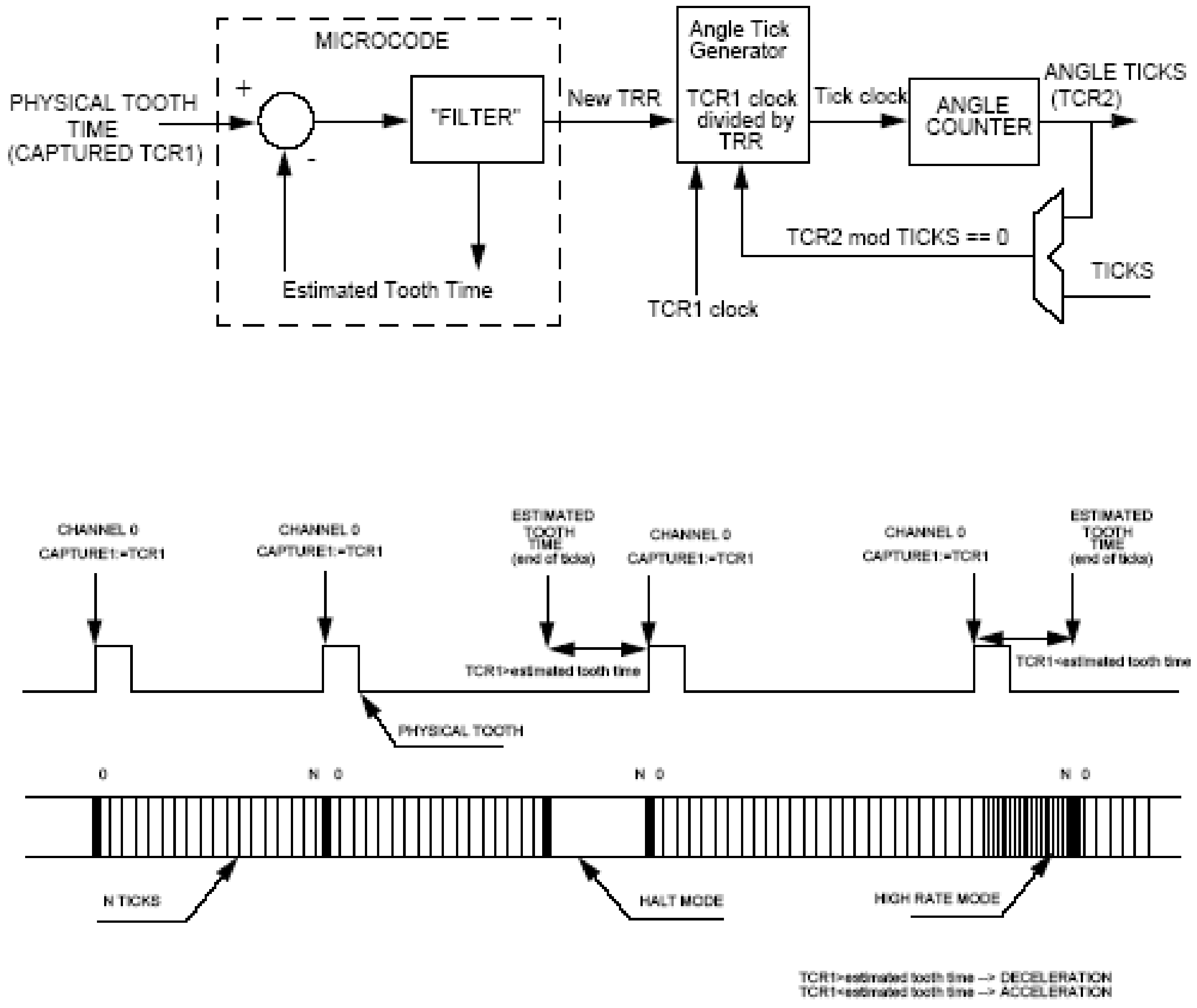

MICROCODE
PHYSICAL TOOTH TIME (CAPTURED TCR1)
+
-
"FILTER"
Estimated Tooth Time
New TRR
Angle Tick Generator
TCR1 clock divided by TRR
TCR1 clock
Tick clock
ANGLE COUNTER
ANGLE TICKS (TCR2)
TCR2 mod TICKS == 0
TICKS
CHANNEL 0 CAPTURE1:=TCR1
ESTIMATED TOOTH TIME (end of ticks)
TCR1>estimated tooth time
TCR1<estimated tooth time
PHYSICAL TOOTH
0
N 0
N TICKS
HALT MODE
HIGH RATE MODE
TCR1>estimated tooth time --> DECELERATION
TCR1<estimated tooth time --> ACCELERATION

Figure 5-12. EAC “PLL”

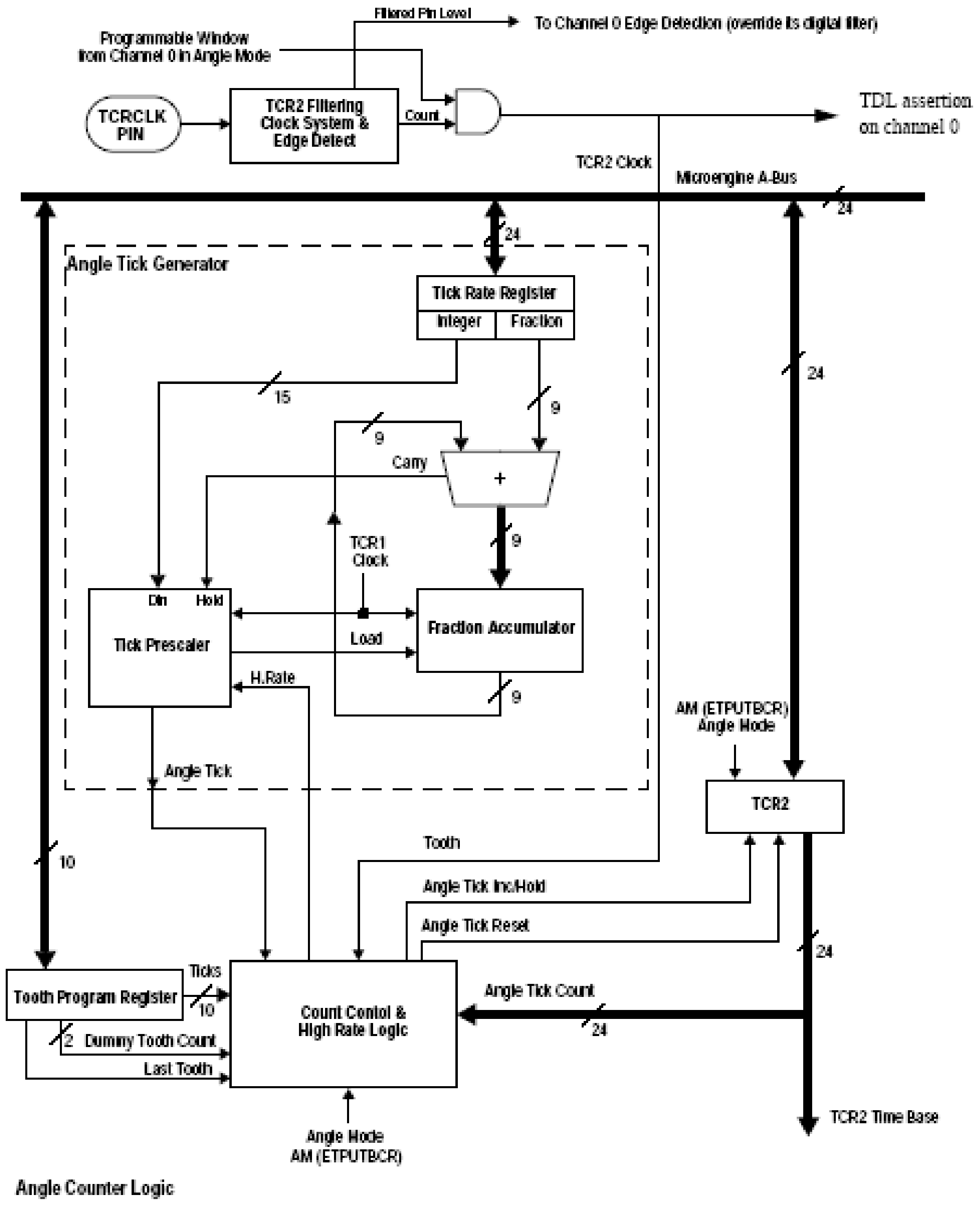


Figure 5-13. eTPU Angle Counter System

5.10.3 Acceleration and Deceleration

Acceleration and deceleration affect the new tooth period relative to the known period of the last tooth. Changes in tooth period may be extreme at very low engine RPM (such as cold start). The worst case of tooth period changes is caused during missing teeth, since there is more time for changes in angular velocity to be unnoticed by the EAC hardware. For example, on cold start (~50 RPM) there may be extreme acceleration: the ratio between a known tooth period before two missing teeth and the new tooth period after the missing teeth can be very high (up to a factor of 75). Acceleration and deceleration effects from tooth to tooth are less extreme as the engine climbs to high RPM.

In case of deceleration, the estimated tooth period ends before the actual tooth detection arrives. In this case, the EAC hardware waits when the tick count has reached TICKS, until the real tooth indication is received, then continues with normal operation. See Table 5-12.

In case of acceleration, the actual tooth period is shorter than the estimated tooth period. As a result, a new physical tooth indication arrives before the end of the estimated tooth period. In this case the EAC closes the gap on High Rate mode by counting on system clock divided by eight to the end of the tooth, advances to the next tooth, and switches back to normal operation mode. See Table 5-12.

The reason that the EAC does not jump directly to the next tooth is the need to provide continuous angle count throughout the whole tooth period, for channels or external STAC bus clients (if TCR2 is a STAC server) which compare angles in an “equal” mode. These peripherals must get all the valid angle values in a sequential manner, to avoid missing angle matches.

TCR2 advancing from one tooth to another is a continuous count, and can be optionally reset at the end of the tooth. An estimated tooth is generated after the Angle Tick Counter reaches the TICKS programmed value.

The EAC works continuously and switches automatically between Normal, Halt and High Rate modes. It relies on the microcode to calculate the estimated tooth period on every tooth, and to update the correct angle tick and tooth parameters in the EAC control registers. On high RPM, tooth period changes are reduced from tooth to tooth, and the EAC may follow the angle with good accuracy for several teeth without microcode intervention.

Software decides whether the EAC handles missing teeth by insertion of “dummy” teeth, or by enlarging the expected tooth period. It is a good practice to locate the flywheel missing teeth in non-critical angles, since a missing tooth may increase the angle measurement error (acceleration and deceleration is detected late).

5.10.4 Angle Tick Generator

The angle tick generator is responsible for generating a programmed number of angle ticks in the tooth period. It generates the ticks in an average rate which ensures completion of the correct number of angle counts in the estimated period of the tooth. Refer to Figure 5-14 for a generic presentation of the angle tick count and the measurement of a single tooth period.

5.10.4.1 Calculating the Angle Tick Period Integer and Fraction

On each tooth the microcode has to update the exact period of a single angle tick used for counting the angle of a tooth. The period of an angle tick or a tooth is measured in units of TCR1 clocks. The microcode can use the eTPU MAC Divider unit, see Section 8.4, “MAC and Divide Unit (MDU),” to divide the tooth period by 1 + the number of angle ticks per tooth, which is stored in the TICKS field of TPR. Refer to Section 5.10.2, “Angle Mode Registers,” for details about the TPR. This division yields the integer part of the angle tick period and the remainder. Dividing again the remainder shifted left nine positions, by the number of angle ticks per tooth translates the remainder to a 9-bit fraction. The microcode concatenates the 15-bit integer and the 9-bit fraction to a 24-bit value and writes it to TRR. The new rate is effective immediately after the next angle tick is generated by the angle tick generator, see Note: .

NOTE

In high-rate mode, the tick keeps being updated at the rate of system clock/8 until it goes back to normal mode, when the new TRR value is used.

For high RPM, note that shifting the tooth period value nine positions to the left prior to the first divide operation would calculate, in one operation, the integer and the fraction. For example: On 60 teeth flywheel running at 1000 RPM, tooth period is 1 millisecond. If TCR1 counts @25 MHz, it counts 25,000 times in a tooth, which can be represented by 15 bits. Therefore the tooth period can be shifted nine positions to the left prior to divide operation, and be represented with 24 bits.

Using shift left nine positions and one divide operation would get the result in MACL register (in MDU) which holds the integer and nine bits of the fraction:

```
Angle_Tick_Rate {Integer[14:0], Fraction[8:0]} = (TCR1ToothPeriod<<9) / Ticks
/* See Note: */
TRR = Angle_Tick_Rate {Integer[14:0], Fraction[8:0]}
```

NOTE

The TCR1ToothPeriod is obtained by microcode by subtracting TCR1 values between two teeth detections. Its comparison with the estimated tooth time indicates acceleration (if minor) or deceleration (if greater) to the microcode.

On low RPM the initial tooth period, measured in TCR1 counts, may be too big to be shifted nine positions to the left. For lower RPM (for example 500 RPM) the tooth period cannot be represented by 15 bits, and shifting it nine positions to the left would lose the MSB. On this case, two divide operations are required as follows: first divide the tooth period by the number of TICKS, i.e. the integer is stored in MACL and remainder in MACH. MACL is stored in another register. MACH is shifted 9 positions to the left and divided again by TICKS. During the second divide, the register which stored the original MACH is shifted left 9 positions. After the divide MACL contains the 9 bits fraction and the other register contains the 15-bit integer, shifted left nine times. The logical OR of the two registers is written to the TRR:

```
Angle_Tick_Rate {Integer[14:0], Remainder[9:0]} = (TCR1ToothPeriod) / Ticks

Angle_Tick_Rate {Fraction[8:0]} = (Remainder[9:0] << 9) / Ticks

TRR = Angle_Tick_Rate {Integer[14:0], Fraction[8:0]}
```

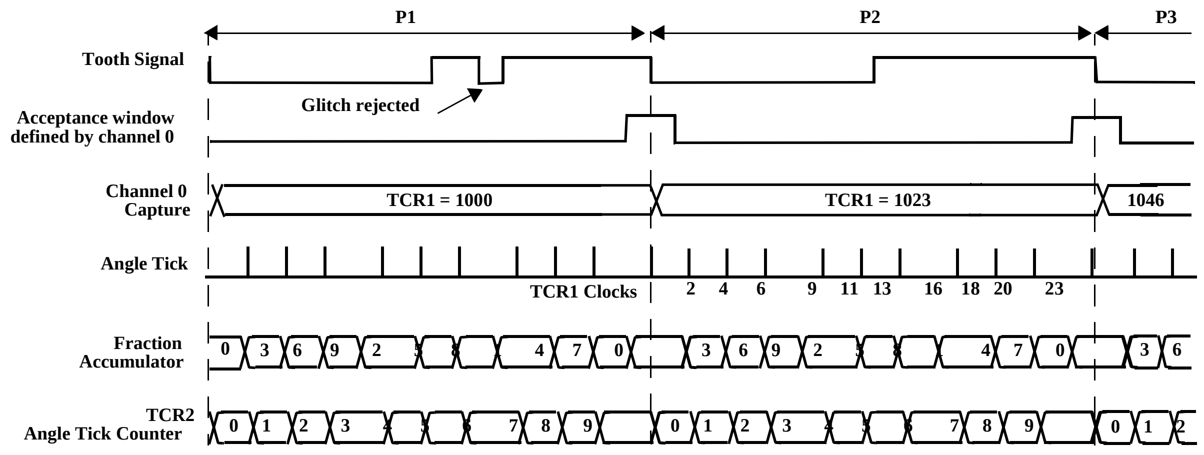


Figure 5-14. Angle Ticks Generation

5.10.4.2 Generating the Angle Ticks

The integer part of TRR is preloaded to a prescaler, which counts down at TCR1 clock rate. When the down counter reaches zero, it generates an angle tick pulse to the Angle Counter Logic and a Load pulse to the Fraction Accumulator. It is then preloaded with most updated TRR integer part. Due to the Load pulse, the 9-bit fraction is accumulated in a 9-bit Fraction Accumulator. If a fraction overflow condition occurs (the 9-bit adder asserts carry out), the accumulator saves the lower 9 bits of the addition result, which is the remaining fractional part. The carry out bit indicates an accumulated integer “one” which means that the angle tick is early by one TCR1 clock. It halts the prescaler operation for one TCR1 clock to compensate the accumulated error generated by the integer prescaler. As a result, the average angle tick period takes into account both the integer and the fraction parts. The accuracy depends on the bit count of the fraction. Using 9-bit fraction part while the width of the field TICKS in register TPR is 10 bits provides accuracy of two LSB on a full scale (TICKS=1023) or one LSB on lower scale (TICKS<=511).

When the Tick Prescaler gets High Rate mode indication from the Angle Counter Logic, it generates angle ticks at a rate of system clock divided by eight. In this case it does not generate Load pulses to the Fraction Accumulator, ignores its “hold” input and preloads internally to a fixed period of eight system clocks. When High Rate mode is entered, the prescaler is preloaded to a period of eight system clocks before its first angle tick generation, ensuring separation of at least eight system clocks between the last Normal mode angle tick and the first High Rate mode angle tick.

5.10.5 Count Control and High Rate Logic

The count control and high rate logic controls TCR2 operation in angle mode, using the angle ticks generated by the angle tick generator. In angle mode, the TCR2 is simply an accumulator of ticks. Count control logic is responsible for advancing, holding and resetting the angle tick counter in the proper timing, such that the TCR2 time base will reflect the correct estimated angle. This logic also includes the tooth program register (TPR), see Section 5.10.2, “Angle Mode Registers,” for more information.

The count control and high rate logic handles deceleration, acceleration, missing teeth and last tooth. On high rate (acceleration) it ensures that the angle bus scans all valid angle values in a rate which can be traced by the STAC interface. This operation enables external STAC clients (if TCR2 is a STAC interface server) or channels working in “equal-only” comparator mode to match the TCR2 exported angle information in “equal” mode, in an exact match.

Because the eTPU channels are capable of capturing either TCR1 or TCR2 due to signal transition, the microcode can get either the angle or time of the physical pin transition. Since channel 0 is connected to the physical tooth, the microcode can get the EAC error in angle domain (tooth appears at the wrong angle) or time domain (physical tooth captured time in channel 0, relative to the estimated tooth time). Note that in angle mode, the transition detect logic of channel 0 is fed from the digital filter of the TCRCLK signal, and not from the channel 0 internal digital filter. This ensures synchronous operation of channel 0 and the EAC hardware.

Another feature of the eTPU channel, when working in single match and single transition enhanced mode, refer to Section 5.8.4.20, “Single Match Enhanced Mode (sm_st_e),” is capturing a single time base due to signal transition before and after the digital filter. This option allows subtracting the digital filter delay to get accurate signal transition timing on the channel. This way, the TCRCLK signal may be programmed with a slow and reliable digital filter, and get accurate time measurement of the digital filter delay.

To assert the end of the estimated tooth period the Count Control and High Rate logic compares the TICKS field in TPR, refer to Section 5.10.2, “Angle Mode Registers,” with the current value of angle tick count. When the Angle Tick value is equal to TICKS, it determines the end of the estimated tooth period. On acceleration this event occurs during High Rate mode operation, after the end of the estimated tooth period. In deceleration, this event occurs during normal mode, before the arrival of a physical tooth. On constant angular velocity, this event appears together with the arrival of a physical tooth.

The following sections describe the operation of the counter control and high rate logic.

5.10.5.1 Normal Mode

In Normal mode the Counter Control logic advances the Angle Counter as if the engine has a constant speed during the tooth period. It receives the angle ticks from the Angle Tick Generator in an average rate which is determined by the Tooth Rate Register (TRR).

When the Angle Counter in TCR2 reaches a multiple of the value stored in TPR field TICKS, the hardware detects the end of the estimated tooth period and advances to the next estimated tooth. If the physical tooth and the estimated tooth arrive at the same time the EAC stays in Normal mode and the angle counter is incremented. If the physical tooth and the estimated tooth do not arrive at the same time, either acceleration or deceleration is detected, and the EAC switches to the proper mode. See Figure 5-15 for a detailed diagram of normal mode behavior.

The microcode which services channel 0 physical tooth transition may update TRR according to various conditions to give the best estimation of the current tooth period, according to the previous tooth period and other engine parameters.

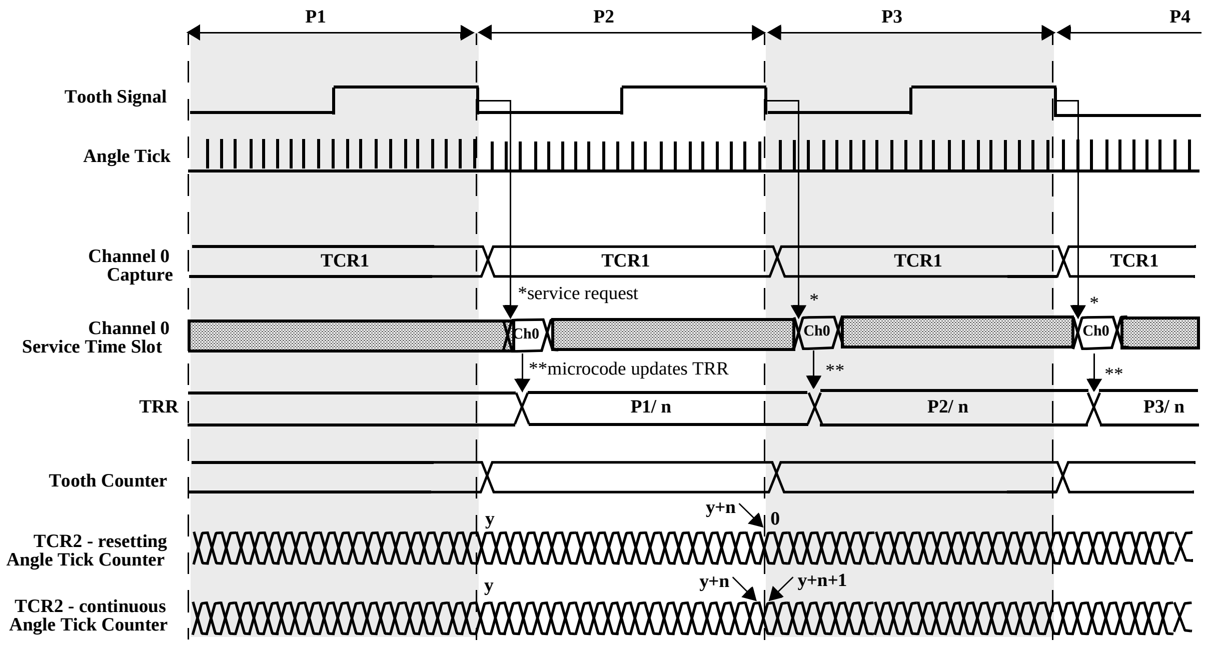


Figure 5-15. Normal Mode

5.10.5.2 Halt Mode (Deceleration)

In case of deceleration, the angle counter reaches the TICKS value before the arrival of the next tooth. The count control logic does not reset the angle counter, neither advances the tooth counter. The count control logic halts the angle counter at the end of the tooth, waiting for the physical tooth to arrive.

When the physical tooth is detected the EAC switches back to normal mode and releases the angle counter to count the angle ticks of the new tooth. Only then the tick counter wraps to 0 and tooth counter is incremented. See Figure 5-16 for a detailed diagram of halt mode behavior.

The microcode service caused by the physical tooth determines the deceleration, calculates the new tooth period and angle tick period and updates TRR. This operation slows the angle tick rate generated by the angle tick generator on-the-fly, to the rate required for the new tooth period.

Since the microcode service is initiated by the physical tooth edge, microcode latency may introduce a small angle error caused by using the TRR value of the previous tooth at the beginning of the current tooth. On high RPM, deceleration is relatively small but the microcode latency may take a significant percentage of the tooth period. On low RPM microcode service latency takes little percentage of the tooth period, but there may be cases of extreme acceleration and deceleration. The microcode latency can be calculated knowing TCR1 value during the service time, and TCR1 value captured in channel 0 due to the physical tooth pin transition. The duration of the halt mode is obtained using the estimated tooth time.

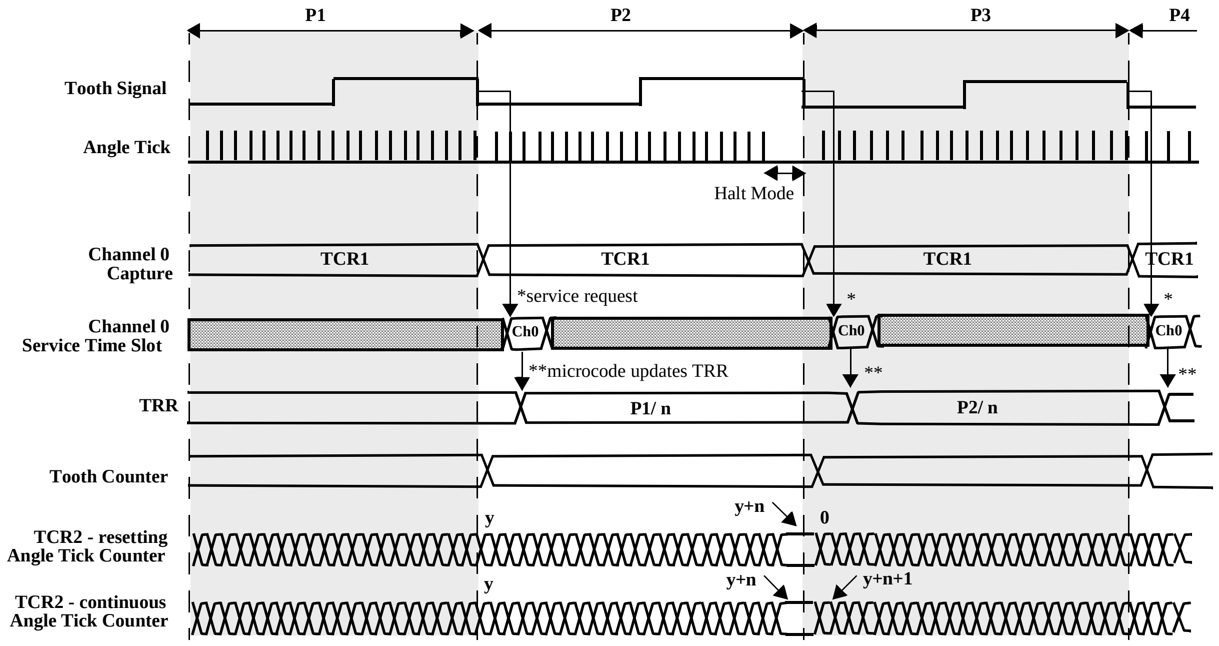


Figure 5-16. Halt Mode, Deceleration

5.10.5.3 High Rate Mode (Acceleration)

In case of acceleration, the next tooth arrives before the angle counter reaches the TICKS value. In this case the high rate logic is responsible for closing the gap and advancing the tooth on the correct timing. The high rate mode operates as follows:

- When the acceleration is detected (physical tooth arrives before the angle counter reaches the TICKS value), the count control and high rate logic switches to high rate mode in which the angle counter counts at rate of system clock divided by eight, until the angle counter reaches the current TICKS value. The TICKS value is sampled in the logic at the beginning of the high rate mode.
- At this point, which represents the estimated tooth edge, the logic advances the angle counter.
- The control logic switches back to normal mode, using the most updated TRR value as input to the angle tick generator. The logic samples the updated TICKS value for the tooth estimation, last tooth indication and number of missing teeth from TPR.

In high rate mode the angle ticks are provided at high speed until the end of the current tooth. This operation is required to scan all the valid angle values of the current tooth, in a rate which is not too high for the STAC interface bus continuous update, but much higher than the rate dictated by TRR.

Channel 0 microcode, which services the physical tooth transition detection, can start its service either before high rate mode operation is complete (the angle counter has not reached the TICKS value) or after the EAC switched back to normal mode. Any physical teeth received while the EAC is in high rate mode must be an error (noise). The received physical teeth are discarded and has no effect on the behavior of the EAC control logic, even if Channel0 is serviced by this transition detection.

At the beginning of high rate mode operation, the TPR value is preloaded into a temporary register in the counter control logic, used for scanning all the valid values to the end of the current tooth, with its appropriate LAST and MISSCNT attributes. This feature is necessary only because the tooth that arrived to start the high rate mode might indicate a need to change LAST or MISSCNT for the next cycle. During High Rate mode, the logic is still completing the last tooth cycle, and requires this data to remain unchanged until the tick count is completed. While the EAC is in high rate mode operation, the effect of microcode update of TPR TICKS field is delayed to the next estimated tooth, after the high rate mode operation is complete. This is because the current physical tooth represents the next estimated tooth. If the microcode updates this field after high rate mode operation is complete, the current physical tooth and estimated tooth are the same, and the effect is immediate. Typically the microcode service may occur during the high rate mode on extreme acceleration situation at low RPM. The microcode operations are always related to the real physical tooth. For correct operation, the TICKS field should not be updated unless the EAC is stopped and re-initialized.

During high rate mode operation, TRR is ignored and the angle tick generator uses system clock divided by eight. Therefore, the TRR update by microcode will take effect only after the EAC switches back to normal mode. If microcode service occurs after the tooth counter has been advanced, the EAC is already back in normal mode, and some angle ticks may have been counted at the rate of the previous tooth. In this case the new TRR value will have immediate effect on the angle tick period, and the microcode should take into consideration the delay from the physical tooth to the estimated tooth in calculation of the next tooth period. See Figure 5-17 for a detailed diagram of high rate mode behavior.

An angle error may be introduced by the duration of the high rate mode. Also, the scheduler latency may introduce a small error by using TRR value of the previous estimated tooth at the beginning of the current tooth. After the estimated tooth has advanced, the duration of the high rate mode operation is the actual delay from the physical tooth edge to the estimated tooth edge. This delay can be obtained by comparing the estimated tooth time with the channel 0 capture register which captured TCR1 on the physical pin transition.

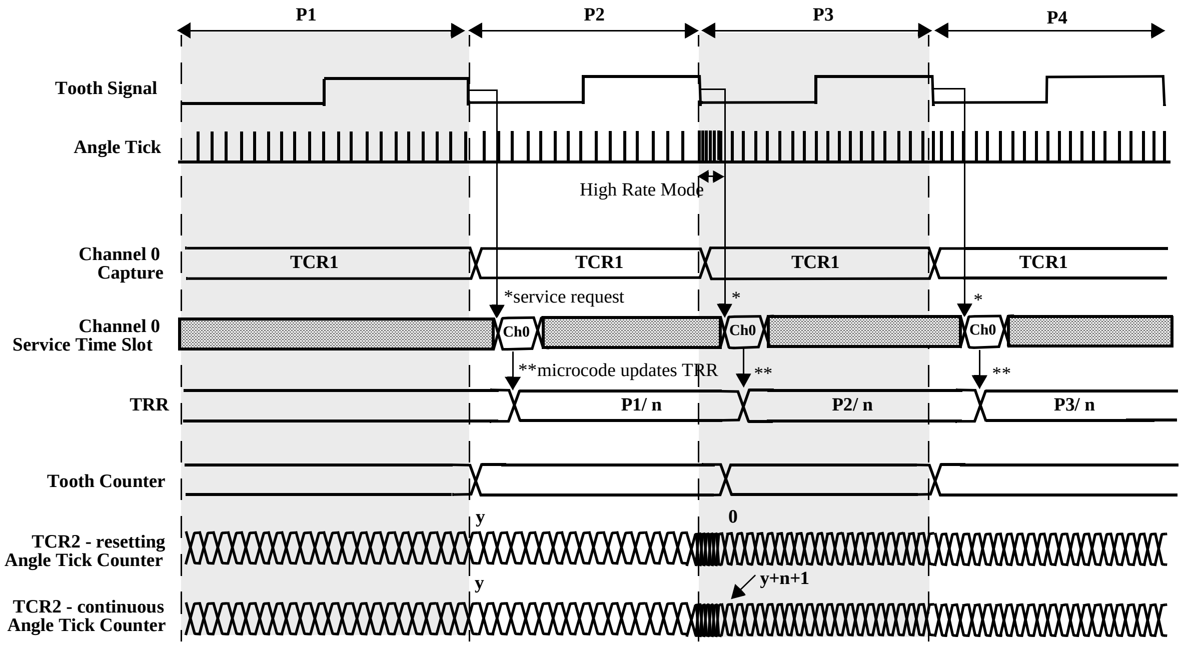


Figure 5-17. High Rate Mode, Acceleration

5.10.6 Special Cases of Missing Teeth and Last Tooth

The EAC handles cases of up to three missing teeth and the last tooth in the engine cycle. The following paragraphs describe these functions.

5.10.6.1 Handling the Last Tooth

The microcode can set the TCR2 counter to work in engine periods (wrap-around count) or continuous angle measurement.

For periodic operation, during the last engine cycle tooth the EAC microcode has to set the LAST flag in TPR. As a result, when the tooth period is ended at the point the EAC receives the next physical tooth edge, the counter control logic generates a reset command to the angle counter in TCR2 instead of an advance command. The operation resets the TCR2 based angle count, indicating a new period of the engine cycle. This implementation provides an engine cycle based periodic angle measurement.

5.10.6.2 Handling Missing Teeth

The EAC can handle up to three missing teeth in two ways:

- Insert a “dummy” tooth instead of the missing tooth, at the estimated point in time. After the “dummy” tooth, the angle tick counter is incremented as if there was a physical tooth. A “dummy” tooth can be inserted during both normal or high rate operation modes. The microcode inserts “dummy” teeth by writing to the MISSCNT field in TPR.
- Count the angle ticks relative to the last physical tooth. The microcode should update the TPR TICKS field to the number of angle ticks included in two, three or four teeth, according to the flywheel type (one, two or three missing teeth). EAC hardware works in its regular manner.

In the first and recommended option, the missing teeth are counted as “regular” teeth by automatic insertion of “dummy” teeth. The microcode has to write a non-zero value to the MISSCNT field in TPR. This field is a 2-bit down counter which affects the operation of the counter control logic.

For example, a toothed wheel with 59 physical teeth (0 – 58) and one missing tooth (59) can be considered as 60 teeth numbered (0 – 59), all having the same number of angle ticks. The microcode has to write “01” to the MISSCNT bits during the period of tooth number 58 to indicate that next tooth (59) is missing.

When the angle tick counter reaches the TICKS value and if MISSCNT is not zero, it is incremented as if a physical tooth has been detected. In addition, the MISSCNT value is decremented to indicate the number of left “dummy teeth” which still need to be generated. Because a dummy tooth was counted, EAC does not enter halt mode and angle tick counter continues incrementing in the absence of a physical tooth detection.

In case of extreme acceleration on very low RPM (cold start) there can be a situation that the first physical tooth after one or two missing teeth appears even before the “dummy” tooth is generated. Due to the acceleration the EAC switches to high rate mode in order to run through all the valid angle values, including the dummy teeth. When the angle counter reaches the TICKS value on high rate mode, and the “dummy tooth” down counter is not zero, the generated “dummy tooth” advances to the next tooth and decrements the “dummy tooth” counter, but does not switch the EAC back to normal mode. The last “dummy tooth” decrements the counter to zero, indicating that no more dummy teeth are to be inserted, and the next tooth is an estimated physical tooth. The EAC continues at high rate mode until the angle tick counter reaches the TICKS value, then advances to the next tooth while switching back to normal mode.

In the second option, the missing tooth is not counted on the angle measurement. For example, a toothed wheel with 59 physical teeth and one missing tooth can be considered as 58 identical teeth numbered (0–57) and tooth number 58 has a double number of angle TICKS. In this case a 720 degrees engine cycle has 118 teeth. TCR2 reflects the real angle, since it counts angle ticks continuously.

5.10.6.3 Combining Missing Teeth and Last Tooth

The last tooth indication takes effect when there are no more missing teeth to be generated, i.e the “dummy tooth” counter value is zero. If, for example, the microcode sets the missing teeth counter to “10” (two missing teeth) and sets the LAST flag, the first and the second dummy teeth will increment the angle counter, and the third estimated tooth, which is the physical tooth (the first of the next cycle), will reset TCR2, because LAST was set. This scheme enables the microcode to define one or more missing teeth to be replaced by “dummy tooth” insertion, and the end of the engine cycle in one service request. It is assumed that the two missing teeth must come together in the same engine cycle, and not split between two engine cycles (either both missing teeth are in one engine cycle or another. The cycle cannot end on a dummy tooth).

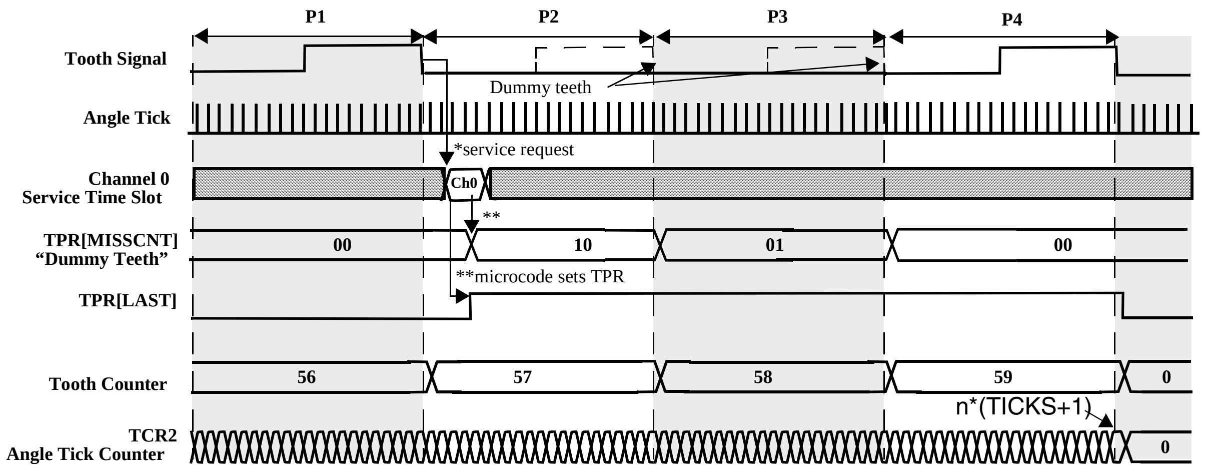


Figure 5-18. Missing Teeth and Last Tooth Combination

5.10.7 Handling Mechanical Tooth Correction

The EAC can handle tooth edge detection errors caused by flywheel mechanical errors. The eTPU application can hold a vector of tooth mechanical errors with one entry per tooth.

This error can be measured in angle ticks which are independent of engine RPM. The TRR can be updated to the fixed period of any tooth, including its mechanical error.

When TCR2, used as a resetting counter, is driving the TCR2 bus, the microcode would fix the angle match values programmed to the eTPU channels according to the vector of tooth mechanical data. If a fixed match angle value exceeds the scope of one tooth due to mechanical error, its match value is transferred to the beginning of the next tooth. This method assumes that the TICKS value is updated on each tooth. Each tooth has a different set of valid values which may complicate the use of TCR2 as angle measurement source.

When TCR2 counts continuously, without being reset, the mechanical correction is practically invisible. Without history, it is impossible to know if tooth period variation is due to misaligned teeth or engine speed variations. Though the tooth has its own programmed TICKS value, TCR2 simply counts angle ticks, loosing the boundary between two adjacent teeth.

5.10.8 Handling Mis-detected Tooth

When a physical tooth signal is missed by the engine sensor, the EAC may get into halt mode at the end of the estimated tooth period, expecting the physical arrival. In this case, a Match time-out event of channel 0 will call service which detects extreme deceleration. The microcode can assert the IPH bit in TPR, to force the detection of the missed physical tooth. It can also calculate the accumulated angle bus error, and fix the next estimated tooth period, to close the gap.

5.10.9 Handling False Tooth Detection

Most false tooth detections, caused by noises on the engine tooth sensor, will be handled by the physical filtering of the tooth signal. False tooth detection that is not filtered can be eliminated by the window blanking filtering, timed by channel 0 match recognitions. The EAC also provides means of fixing false detection of an additional tooth which passed the window filter. When such an event occurs, the EAC switched to high rate mode (advancing to the next tooth) and when the next physical tooth arrives, an extreme acceleration is detected; the EAC sees the remaining portion of the current tooth period as another tooth period.

NOTE

If the masking is setup right, there are only rare cases where the EAC will be incorrect for more than one tooth.

The microcode can detect the situation when the acceleration is not realistic, or when immediately after the detection of this extreme acceleration, the following tooth indicates extreme deceleration back to the original RPM.

When the microcode detects such a case, the tooth counter has been advanced by mistake to the next tooth. The microcode can set the HOLD bit in the TPR, forcing the EAC to halt and wait for the next physical tooth to close the gap. When the next physical tooth arrives, HOLD is automatically negated and the EAC proceeds from that point to the remaining portion of the tooth period.

Chapter 6 Scheduler

Every function is composed of one or more threads. A thread consists of a group of instructions that, once execution begins, cannot be interrupted by host or channel events (except forced end). Active channels need to be serviced and are granted time by the scheduler for thread execution. Since one microengine handles several channels operating concurrently, the function threads must be executed serially.

The task of the scheduler is to recognize and prioritize the channels needing service and to grant execution time to each channel. The time given to an individual thread for execution or service is called a time slot. The duration of a time slot is determined by the number of instructions executed in the thread plus SPRAM wait-states received, and varies in length.

At any time, an arbitrary number of channels can require service. To request service, channel logic, eTPU microcode or the host application notifies the scheduler by issuing a service request.

6.1 Channel Enabling and Priority Assignment

Every channel is assigned one of three priority levels: high, middle, or low by the host CPU, through the channel configuration register field CPR, see Section 4.6.1, “eTPU Channel x Configuration Register (ETPUCxCR).” These registers are also used to disable the channel, which is equivalent to assigning it a “null” priority. In this case, the scheduler does not grant any of its service requests.

It is possible to change the channel priority level or disable it dynamically. If the host disables a channel when it is currently being serviced, channel service thread will complete. This means that it is possible for the output level of a channel signal to change, or a host interrupt occur, even after its priority register was written to “null”. For instance, if an output transition is scheduled, the transition will occur even after the channel is disabled.

Service requests previously pending or that occur while a channel is disabled remain asserted while the channel is disabled, and are serviced if the channel is enabled again, in due time determined by the priority scheme and concurrent requests from other channels. Channels are disabled after reset, and it is recommended to configure a host service request for initialization of a channel before that channel is enabled to active priority, see Chapter 12, “Initialization/Application Information.”

NOTE

Reset also clears the channel service requests but any link service requests made by channels already initialized could be granted if the priority were set to a non-zero value before the HSR is written.

6.2 Channel Priority Schemes

The scheduler holds a service grant register with one bit for each channel. Once the scheduler grants a time slot to channel, the service grant bit for that channel is asserted in the service grant register. When all channels in a same priority level are serviced, their service grant bits are cleared at the end of the thread. When the service grant bit of a channel is set, the channel may request new service but is not serviced again before its service grant bit is cleared.

Priority level is determined by the value of the priority field and should be set to a level to meet latency requirements. A channel having a function that requires the most frequent or more immediate service should be allocated a high priority level.

The eTPU employs a primary and a secondary priority scheme. These two schemes ensure frequent servicing of high-priority channels and guarantee a time slot for all channels requesting service, regardless of their priority level. The primary scheme prioritizes requesting channels that have different priority levels; the secondary scheme prioritizes requesting channels that have the same priority level.

Initially, a channel requests service and is granted a time slot by the scheduler. The service grant bit is asserted. If only high-level channels constantly receive service first because of their priority level, middle- and low-level channels have no guarantee of being serviced, i.e., the middle- and low-level channels would only be serviced if no high-level channels request service. To ensure that each priority level receives an opportunity for servicing, every time slot has a fixed priority level that the scheduler honors first. Divided into sets of seven, time slots are numbered from one to seven. Figure 6-1 illustrates the numbered time slots in sets of seven (fields A and B) and identifies their assigned default priority level. The high level has more time slots than the middle and low levels. Out of every seven time slots available, four are assigned to honor high-level channels first, two are assigned to honor middle-level channels first, and one is assigned to honor low-level channels first. Only one request (in each engine) is serviced per time slot.

NOTE

Each engine has a separate and independent scheduler, and no action in eTPU_A affects the operation of the scheduler in eTPU_B.

When no channel requests service and the microengine is idle the priority scheme is initialized to time slot one.

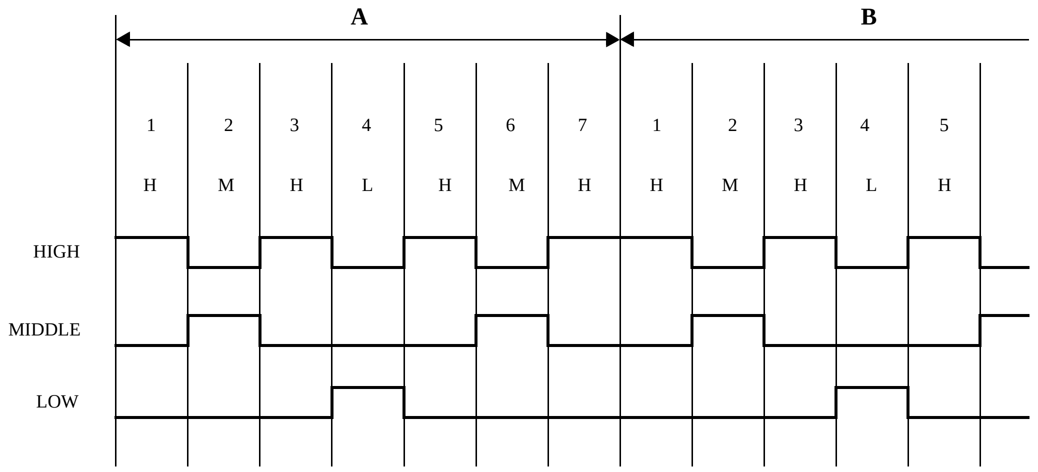


Figure 6-1. Time Slot Priority levels

6.2.1 Primary Scheme: Priority Among Channels on Different Levels

Although time slot priority assignment is fixed, the servicing priority is not. The primary scheme acknowledges the priority level assigned to a time slot, granting service first to a channel having the same priority. In Figure 6-1, time slot one has a high-level assignment; therefore, a high-level channel requesting service is recognized first. However, if no high-level channel requests service, the scheduler recognizes a requesting middle-level channel. If this level has no request, the scheduler continues to the low-level. If no requests occur, the scheduler truncates the cycle and starts a new cycle at time slot one, waiting for the first request. Granting service to a different-level channel is called priority passing. The order of passing always gives the highest priority to the assigned level, and the second priority to the higher of the remaining requesting priority levels as shown in Table 6-1.

Table 6-1. Priority Passing

Assigned Priority Level	Next Priority Level	Next Priority Level
High	$\rightarrow$ Middle	$\rightarrow$ Low
Middle	$\rightarrow$ High	$\rightarrow$ Low
Low	$\rightarrow$ High	$\rightarrow$ Middle

When priority is passed to another level, that level is serviced and the fixed-priority-level sequence is resumed with the next time slot.

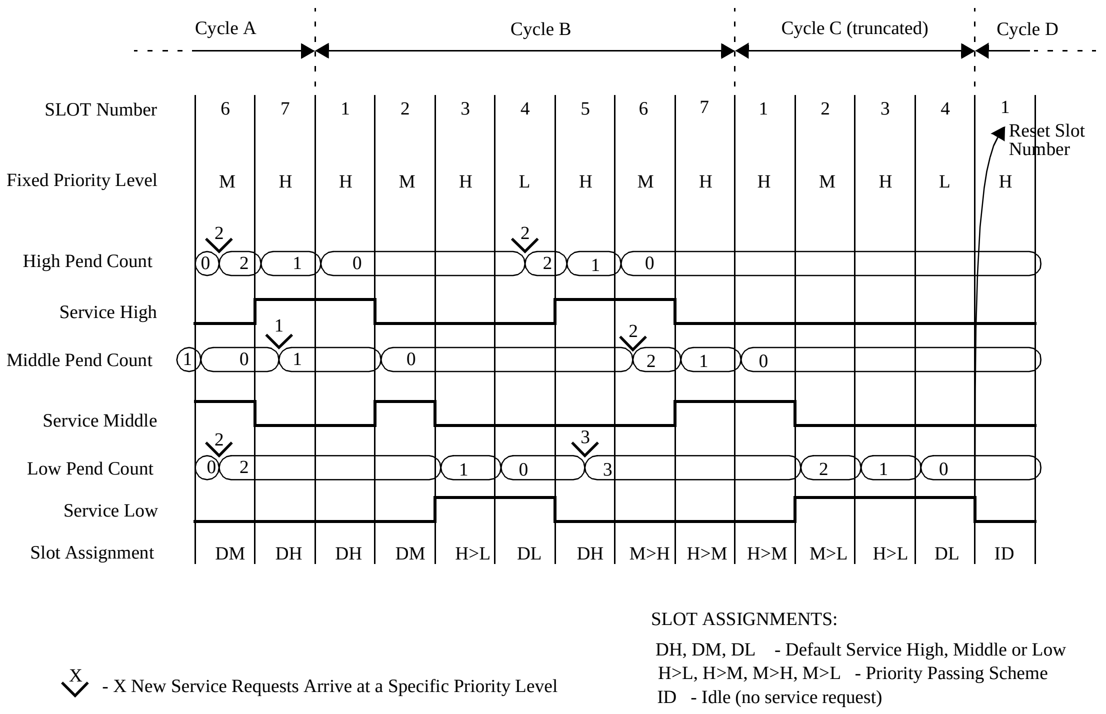

Cycle A
Cycle B
Cycle C (truncated)
Cycle D
SLOT Number
Fixed Priority Level
High Pend Count
Service High
Middle Pend Count
Service Middle
Low Pend Count
Service Low
Slot Assignment
Reset Slot Number
DM DH DH DM H>L DL DH M>H H>M H>M M>L H>L DL ID
SLOT ASSIGNMENTS:
DH, DM, DL - Default Service High, Middle or Low
H>L, H>M, M>H, M>L - Priority Passing Scheme
ID - Idle (no service request)
X - X New Service Requests Arrive at a Specific Priority Level

Figure 6-2. Priority Passing Example

Examples of priority passing are shown in Figure 6-2. Each cycle contains seven time slots (or less if no service request exist and the cycle resets to time slot 1). In cycle B, no high-level or middle-level service requests are present before time slot three which is assigned by default to high-level priority. Thus, time slot three is passed to the low level. In cycle B there are also no middle-level service requests before time slot six, so it passes the priority to a requesting high-level channel. During time slot six no more high level requests are left, but two new middle-level requests arrive, and there are also three low level pending service requests. Thus, time slot seven of cycle B and time slot one of cycle C are passed to the middle-level which is the next priority level after high. Time slots two and three of cycle C are passed to the low level which contains the three remaining channel service requests. At time slot four of cycle C the last low level request is serviced, and the scheduler passes to idle state. At this point the cycle C is truncated and the scheduler passes to time slot one of cycle D.

6.2.2 Secondary Scheme: Priority Among Channels on the Same Level

Because channels can randomly request service, channels having the same priority level will inevitably request service simultaneously. A secondary scheme prioritizes these requests. The scheduler services channels on each of the three priority levels, beginning with the lowest numbered channel on that level.

6.2.3 Priority Scheme Example

The overall priority scheme simultaneously incorporates both primary and secondary schemes. Combining both schemes in the following example conveys their correlation.

1. One high-priority and one low priority channels request service, while the scheduler is in time slot one. Having its service request bit asserted, a single high-level channel is granted the time slot, which has high-level priority (primary scheme) and its service grant bit is asserted. At the end of the thread, the service grant bit is negated (no more requests of high priority level channels).
2. The scheduler proceeds to time slot two, which has middle-level priority; however, no middle-level channel is requesting service. Priority is passed to the high level, but no high-level channel is requesting service; therefore, priority is passed again, and service is granted to the single requesting low-level channel. Once serviced, this channel's grant bit is negated (no more low-level requests).
3. The scheduler resumes with the fixed-priority sequence on time slot three; however, no channels are requesting service. The scheduler returns to time slot one, waiting for requests.
4. Two high-level and two middle-level channels simultaneously request service. Being in time slot one which is assigned high priority, the scheduler finds the lowest numbered high-level channel (secondary scheme) and selects it for service. This channel's service grant bit is asserted.
5. The scheduler continues to time slot two, which has middle priority (primary scheme), and allocates the slot to the lowest numbered middle-level channel requesting service (secondary scheme). The scheduler notes the still unserviced middle-level channel and proceeds to time slot three.
6. Time slot three is allocated for high priority. The slot is allocated to the remaining unserviced high-priority channel, and the channel's service grant bit is asserted. The scheduler checks again at the end of the thread. All service grant bits of high-level requested channels are asserted; therefore, all high-priority channels that requested have been allocated execution time. Under this condition, all service grant bits of the high-level serviced channels are negated. The scheduler proceeds to time slot four.
7. Time slot four is allocated for low-priority channel; however, no low-level channel is requesting service. Priority is passed to the high level, but no high-level channel is requesting service; therefore, priority is passed again, and service is granted to the remaining middle-level channel which requests service. This channel's service grant bit is asserted. The scheduler checks again at the end of the thread. All grant bits of middle-level requested channels are asserted; therefore, all middle-priority channels have been allocated execution time. Under this condition, all service grant

bits of the middle-level serviced channels are negated. The scheduler proceeds to time slot five. Before the scheduler transitions to the next time slot, a new request for service is received from a low priority channel.

8. Time slot five is allocated for high-priority channels, but there are no more requests from high-priority or middle priority channels. The single low-level channel which required service is granted time slot five. Once serviced, the channel’s service grant bit is asserted. Next, the service grant bit is negated (no more requests of low priority level channels).
9. The scheduler resumes with the fixed-priority sequence on time slot six; however, no channels are requesting service. The scheduler returns to time slot one and waits for requests.

6.3 Time Slot Latency

Latency is the amount of time between a service request and the beginning of service on that channel. The following factors affect latency:

- Number of active channels
- Number of channels on a priority level
- Number of available time slots on a priority level
- Number of microcycles required to execute a thread of a function
- Number of parameter RAM accesses collisions during execution of a function thread
- System clock frequency.

Each time slot may require a different number of microcycles, depending on the thread of a function to be executed. This variation is shown in Figure 6-3.

For more details on latency evaluation, see Section 12.5, “Estimating Worst Case Latency.”

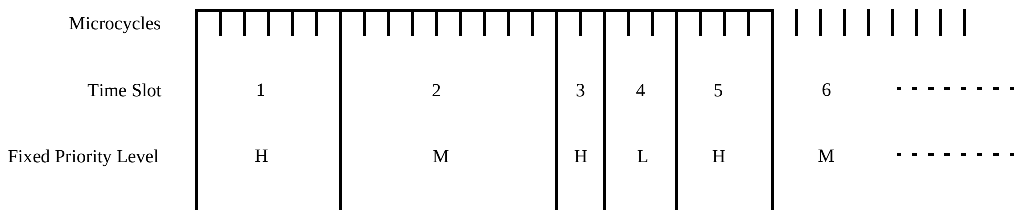


Figure 6-3. Time-Slot Variation

Chapter 7 Functions and Threads

7.1 Introduction

eTPU processing is event-driven, in the sense that eTPU microcode only runs to service a request from an event. Service requests may result from the occurrence of any of the following events:

- the host CPU writing a non-zero value to the channel HSR (host service request) field in ETPUCxHSR register.
- a time base match, an input signal transition, or a specific combinationn of them (depending on the channel mode currently configured).
- a link service request.

A given event is always associated to only one channel:

- there is one HSR register field for each channel
- each signal is associated with only one channel, which has its own registers and independent mode configuration.
- each link service request can have only one channel as a target.

Service request processing is done by a set of microengine routines. A set of related routines that implement a specific channel application is called a function. One or more functions reside on SCM, limited only by the SCM space available, size of microcode functions and the number of entry points available. Each engine can execute up to 32 functions (one at a time).

A function can be assigned to several channels, but only one function can be assigned to a given channel at a time. This is defined by the host through the channel configuration registers, see Section 4.6, “Channel Configuration and Control Registers.”

The term “thread” will be used hereafter to refer to a service routine of a function, or its execution. A thread is constructed of a number of microinstructions, typically the code necessary to set up the channel logic to detect the next input transition or control the next timed event. Once a thread begins, its execution cannot be interrupted by another function. Execution can be halted by the host. A thread finishes when an END microinstruction is executed.

A given thread is selected and called by the scheduler depending on the following:

- the type of event that generated the service request.
- the function assigned to the target channel.
- target channel pin state.
- the state of the channel logic.
- the priority assigned to the target channel, relative to the priorities of other channels with pending service requests

The mechanism to select a thread based on the channel function and type of event is described in the Section 7.2, “Entry Points.”

The priority mechanism that determines the order of thread execution amongst pending service requests is described in Section 5.3, “Scheduler.”

7.2 Entry Points

7.2.1 Entry Table

Each thread has its own entry point. An entry point contains the SCM address of the thread’s first instruction. Since the entry point is determined by channel conditions, the thread selected is unique to these conditions. Therefore, the thread is “aware” of the channel conditions at the time of the thread’s selection. For a complete entry point description, see Section 7.2.5, “Entry Point Format.”

Once the scheduler chooses a channel among pending service requests, the entry point is selected from an entry table, based on the function assigned for the channel and other conditions. Entry table layout is shown in Figure 7-1.

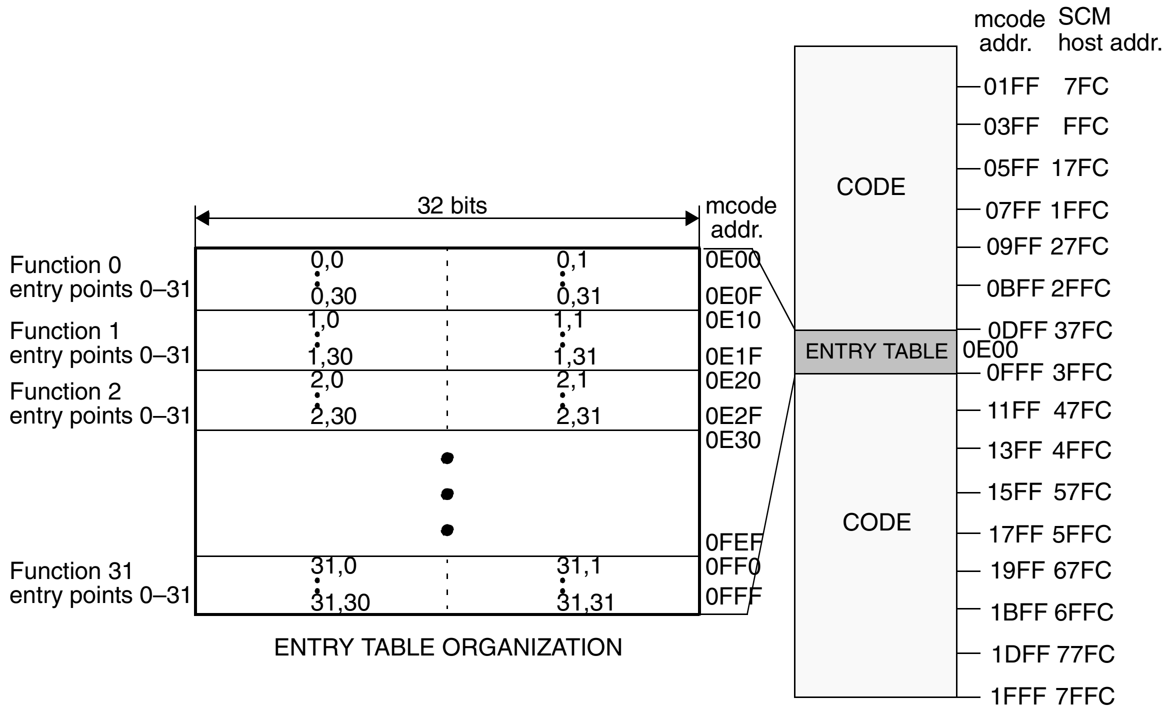


Figure 7-1. Entry Table

The entry table is organized by functions. Each function can have up to 32 entry points of 16 bits each, corresponding to 32 possible threads per function. Each entry point location in the table corresponds to a combination of events and channel states see Section 7.2.2, “Entry Point Address Generation.” A single thread can be associated to more than one combination, having its entry point repeated in the table. Each 32-bit word in the entry table holds two entry points.

The entry table can be placed in any SCM address multiple of the entry table maximum size, determined by the field ETB[4:0] in the ETPUECR register, Table 4-4. However, it is recommended to place the entry table at the start of the SCM to get continuous code memory and to ease the eventual migration of the code from larger parts down to smaller ones without rearranging the binary image, but this is not a restriction. Unused entry points locations may be used for microcode, so this organization extends the microcode continuous area to the unused area of the entry table. For this purpose, function numbers should be selected from 0 up to 31. If, for example, only 8 functions are implemented, only the entry table locations for functions 0 to 7 are used, and the entry table locations for functions 8 to 31 can be used as microinstruction memory (adding extra continuous 1536 bytes for microprogram usage).

One way of implementing different sets of functions is having more than one entry table, and configuring the eTPU with the appropriate one for the application by changing ETPUECR register field ETB. Note that the engines can use different entry tables, with or without the same set of functions.

7.2.2 Entry Point Address Generation

The entry point address within the entry table is determined by the function assigned to the channel, the state of the channel, the type of event, and the condition encoding scheme. Together with the entry table base address, they form the entry point address at the SCM, as shown in Figure 7-2.

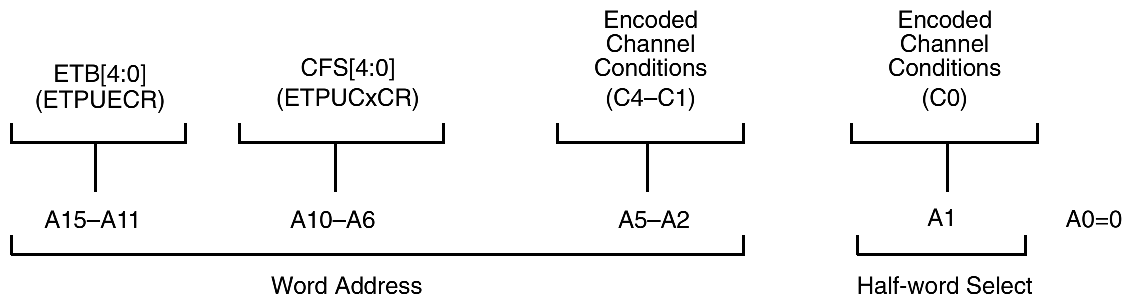


Figure 7-2. Entry Point Address (Host Address Offset)

The type of event and channel state are coded in the encoded channel conditions field C[4:0], according to one of two encoding schemes:

- Standard entry table condition encoding scheme, shown in Table 7-1, which gives priority to host service requests.
- Alternate entry table condition encoding scheme, shown in Table 7-2, which focuses on other events and state decoding.

The events that cause service requests contribute the to encoding of the entry point. These events have four origins:

1. Match Event.

   A match is caused by greater/equal match, or equal-only, between the value TCR1/2 and the value stored in the channel match registers. eTPU channels support single and double match in various modes of match recognition; see Section 5.5.2, “Match Recognition,” for more information on match recognition.

2. Transition event.

   A transition is a detection of a specified transition edge for a channel input signal. The eTPU channels support single and double transition, which together with the double match options provide various modes of transition detection; see Section 5.5.3, “Transition Detection and Time Base Capture,” for more information on transitions.

3. Channel link.

   A channel linking service request occurs when the microcode writes a channel number to the LINK register. Link service request allows one channel to activate another; see Section 5.5.5, “Channel Link,” for more information on channel requests.

4. Host service request.

   A host service request is when the host writes a non-zero value to the HSR bits of the channel. For more information, see Section 5.2.5, “Host Service Requests.”

NOTE

If the match or transition service requests are inhibited, then no service request will be issued by those events. However, if another service request is issued, the event flags (MRL_A, MRL_B, TDL_A, and TDL_B) may further qualify the channel condition encoding.

The columns in Table 7-1 and Table 7-2 illustrate how the host request bits, link request, Match1/Trans2, and Match2/Trans1 determine the type of event. A “1” represents a condition which must be present to recognize the event, while a “0” represents a condition which must not be present. An “x” indicates that the condition is not considered. Note that match and transition events may occur and not be recognized, and in this case it assumes value 0 for the condition encoding. The recognition of an occurred event depends on the channel mode assigned and other conditions, as described in Section 5.5, “Enhanced Channels.”

The host service request bits column refers to the value written by the host CPU to the host service request register (ETPUCxHSRR) of the channel being serviced. Note that the bits on this row are coded (3-bit representation). If the value of HSR is not zero, then the host actually requested service.

The link request column refers to the occurrence of a channel link request.

The Match1/Trans2 column refers to the recognition of either a match event specified by Match1 channel register or the detection of a channel input signal event specified by the IPAC2 configuration register, see Section 5.5.1.2, “Pin Control Registers.”

The Match2/Trans1 column refers to the recognition of either a match event specified by Match2 channel register or the detection of a channel input signal event specified by the IPAC1 configuration register, see Section 5.5.1.2, “Pin Control Registers.”

NOTE

There are no transition detections if a channel is used for output only, so the Match2/Trans1 columns in Table 7-1 and Table 7-2 simply represent Match2 in this case. Also the Match1/Trans2 columns would only represent Match1 if a channel is used for output only.

Besides those events, the following channel state conditions help to determine the entry point:

1. Channel flags 0 and 1: these are channel-internal flags (not in SPRAM) associated with a channel. Their values are set by microcode, see Section 5.9.3.1, “Channel Flags Operations.”
2. Input pin state: the state (0 or 1) of the channel input signal after the enhanced filter, see Section 5.5.6, “Enhanced Digital Filter (EDF).”

The two entry table condition encoding schemes combine events and state conditions differently, as detailed in following sections.

7.2.3 Standard Condition Encoding Scheme

In this scheme, shown in Table 7-1, all 7 HSR combinations are used and other event type columns are marked “x” when HSR is non-zero, indicating that host service request has priority over any other type of event. However, when an HSR service thread is called (entry numbers 0 to 9), other events may also have been recognized, and it is microcode’s responsibility to check them.

When HSR is 0, i.e., the host did not issue a service request to the channel, the other event conditions, the input signal state and channel flags determine the entry point. Note that the channel flag 1 does not influence the encoding in this scheme.

Table 7-1. Standard Channel Condition Encoding Scheme

No.	Encoded Channel Conditions [C4:C0]	Host Service Request Bits	Link Request	Match 1 / Trans.2	Match.2 / Trans.1	Input Pin State	Channel Flag1[1]	Channel Flag0
0	00000	001	x	x	x	0	x	0
1	00001	001	x	x	x	0	x	1
2	00010	001	x	x	x	1	x	0
3	00011	001	x	x	x	1	x	1
4	00100	010	x	x	x	x	x	x
5	00101	011	x	x	x	x	x	x
6	00110	100	x	x	x	x	x	x
7	00111	101	x	x	x	x	x	x
8	01000	110	x	x	x	x	x	x
9	01001	111	x	x	x	x	x	x
10	01010	000	1	1	1	x	x	0
11	01011	000	1	1	1	x	x	1
12	01100	000	0	0	1	0	x	0

Table 7-1. Standard Channel Condition Encoding Scheme

No.	Encoded Channel Conditions [C4:C0]	Host Service Request Bits	Link Request	Match 1 / Trans.2	Match.2 / Trans.1	Input Pin State	Channel Flag1[1]	Channel Flag0
13	01101	000	0	0	1	0	x	1
14	01110	000	0	0	1	1	x	0
15	01111	000	0	0	1	1	x	1
16	10000	000	0	1	0	0	x	0
17	10001	000	0	1	0	0	x	1
18	10010	000	0	1	0	1	x	0
19	10011	000	0	1	0	1	x	1
20	10100	000	0	1	1	0	x	0
21	10101	000	0	1	1	0	x	1
22	10110	000	0	1	1	1	x	0
23	10111	000	0	1	1	1	x	1
24	11000	000	1	0	0	0	x	0
25	11001	000	1	0	0	0	x	1
26	11010	000	1	0	0	1	x	0
27	11011	000	1	0	0	1	x	1
28	11100	000	1	0	1	x	x	0
29	11101	000	1	0	1	x	x	1
30	11110	000	1	1	0	x	x	0
31	11111	000	1	1	0	x	x	1

▨ Host Service Request

1. The channel flag 1 does not influence the encoding in the standard channel condition encoding scheme.

7.2.4 Alternate Condition Encoding Scheme

This scheme is shown in Table 7-2. Because the HSR bits cannot be tested by microcode, only three distinct categories are allotted to specific host service request bit combinations:

1. HSR=010 or 011, which are coded into the same entry points (0 to 3)
2. HSR=100,101 or 001, which are all coded into entry point 4
3. HSR=110 or 111, which are both coded into entry point 5

The remaining entry points use both channel flags for better state decoding, making this scheme better suited for functions which need more states and/or faster state decoding, without needing many HSRs.

Table 7-2. Alternate Channel Condition Encoding Scheme

No.	Encoded Channel Conditions [C4:C0]	Host Request Bits	Link Request	Match 1 / Trans.2	Match.2 / Trans.1	Input Pin State	Channel Flag1	Channel Flag0
0	00000	01x	x	x	x	0	x	0
1	00001	01x	x	x	x	0	x	1
2	00010	01x	x	x	x	1	x	0
3	00011	01x	x	x	x	1	x	1
4	00100	10x/001	x	x	x	x	x	x
5	00101	11x	x	x	x	x	x	x
6	00110	000	1	0	0	0	x	x
7	00111	000	1	0	0	1	x	x
8	01000	000	x	1	0	0	0	0
9	01001	000	x	1	0	0	0	1
10	01010	000	x	1	0	0	1	0
11	01011	000	x	1	0	0	1	1
12	01100	000	x	1	0	1	0	0
13	01101	000	x	1	0	1	0	1
14	01110	000	x	1	0	1	1	0
15	01111	000	x	1	0	1	1	1
16	10000	000	x	0	1	0	0	0
17	10001	000	x	0	1	0	0	1
18	10010	000	x	0	1	0	1	0
19	10011	000	x	0	1	0	1	1
20	10100	000	x	0	1	1	0	0
21	10101	000	x	0	1	1	0	1
22	10110	000	x	0	1	1	1	0
23	10111	000	x	0	1	1	1	1
24	11000	000	x	1	1	0	0	0
25	11001	000	x	1	1	0	0	1
26	11010	000	x	1	1	0	1	0
27	11011	000	x	1	1	0	1	1
28	11100	000	x	1	1	1	0	0

Table 7-2. Alternate Channel Condition Encoding Scheme

No.	Encoded Channel Conditions [C4:C0]	Host Request Bits	Link Request	Match 1 / Trans.2	Match.2 / Trans.1	Input Pin State	Channel Flag1	Channel Flag0
29	11101	000	x	1	1	1	0	1
30	11110	000	x	1	1	1	1	0
31	11111	000	x	1	1	1	1	1

▨ Host Service Request

7.2.5 Entry Point Format

The entry point format is illustrated in Figure 7-3. Entry point information includes a preload-parameter selection field, a match enable field, and the first address (word format) of the thread.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PP	ME	MICROCODE ADDRESS													

Figure 7-3. Entry Point Format

Table 7-3. Entry Point Format

Bits	Name	Description
15	PP	Preload Parameter. PP indicates which pair of channel parameters are loaded into registers P and DIOB from the SPRAM prior to the execution of a thread. Preloading occurs during the time-slot transition period. 1 microengine register P is preloaded from parameter 2 and DIOB from parameter 3. 0 microengine register P is preloaded from parameter 0 and DIOB from parameter 1. The parameter numbers are offsets from the channel parameter base address. For more information on parameters, see Section 5.2.3, “Parameter Access.”

Table 7-3. Entry Point Format

Bits	Name	Description
14	ME	Match Enable. ME specifies whether match event recognitions are enabled or disabled for the thread specified by the MICROCODE ADDRESS field. If matches are disabled, match recognition can only occur after channel service. For more details refer to 4.5.2 Match Recognition. 1 Matches are enabled during the thread. 0 Matches are disabled during the thread. The disabling of Match1/2 recognition by MEF is dependent on IPAC1/2 configuration on the serviced channel. If IPAC1=1xx, Match1 is not disabled by ME=0. Likewise, IPAC2=1xx overrides the effect of ME on Match2 to "always on" If IPAC1/2=0xx. See Section 5.5.1.2, “Pin Control Registers,” and Section 7.3, “Time Slot Transition,” for more details.
13–0	MICROCODE ADDRESS	This field specifies the microcode address on which the thread is to begin execution.

7.3 Time Slot Transition

The time slot transition period (also called TST for short) is the interval just before the start of a thread, between the granting of a service request by the scheduler and execution of the first microinstruction., during which all channel-specific context is loaded for the new serviced channel. The primary tasks completed during this period include:

- Reset the match enable flag (MEF) during the first two microcycles.
- Update of the CHAN register with the number of the new channel to be serviced.
- Parallel update of ERT_A and ERT_B registers from Capture1 and Capture2 registers of the new serviced channel.
- Sampling of the branch conditions of the new channel to be serviced into the branch logic (this means flags TDL_A/B, MRL_A/B, LSR, FM[1], FM[0], PSS and PST). The branch conditions are coherent with the captured time bases (if MRL_A/B, TDL_A/B are set at the same time of the sampling, either both old flag state and capture values are sampled, or both new values are sampled).
- Formation of the entry point address.
- Copy the ME bit in the entry point into MEF.
- Access to the entry point location and getting the first microinstruction address.
- Preload of two parameters from the SPRAM into registers P (32 bits) and DIOB (24 bits).
- Fetch the first instruction of the thread to be executed for the new channel.
- Preset the RAR register value, see Section 5.8.1.7, “Return Address Register (RAR).”

The preload operation is 32 bits wide for P and 24 bits wide for DIOB. The P register is loaded with the whole 32-bit parameter. The DIOB register is loaded with the lower 24 bits of the parameter. Preload of P-DIOB pair of parameters is atomic with respect to host and CDC accesses. For more details see Section 5.4, “Parameter Sharing and Coherency.”

The engine where the time slot transition period occurs does not execute any instructions during TST, but the other engine can execute normally. Match1/2 is unconditionally disabled at the first two TST microcycles, if IPAC1/2=0xx (respectively). After the first two cycles, match recognition can be disabled or not, depending on IPAC1/2 field and ME. For more details, see Section 5.5.2, “Match Recognition.”

A time slot transition takes a minimum of 3 microcycles (6 system clocks), which may be extended due to SPRAM arbitration wait-states for the first preload access, see Section 5.4.5, “SPRAM Arbitration.”

The value of any register other than P, DIOB, CHAN, ERT_A, ERT_B and RAR is not guaranteed at the beginning of the thread.

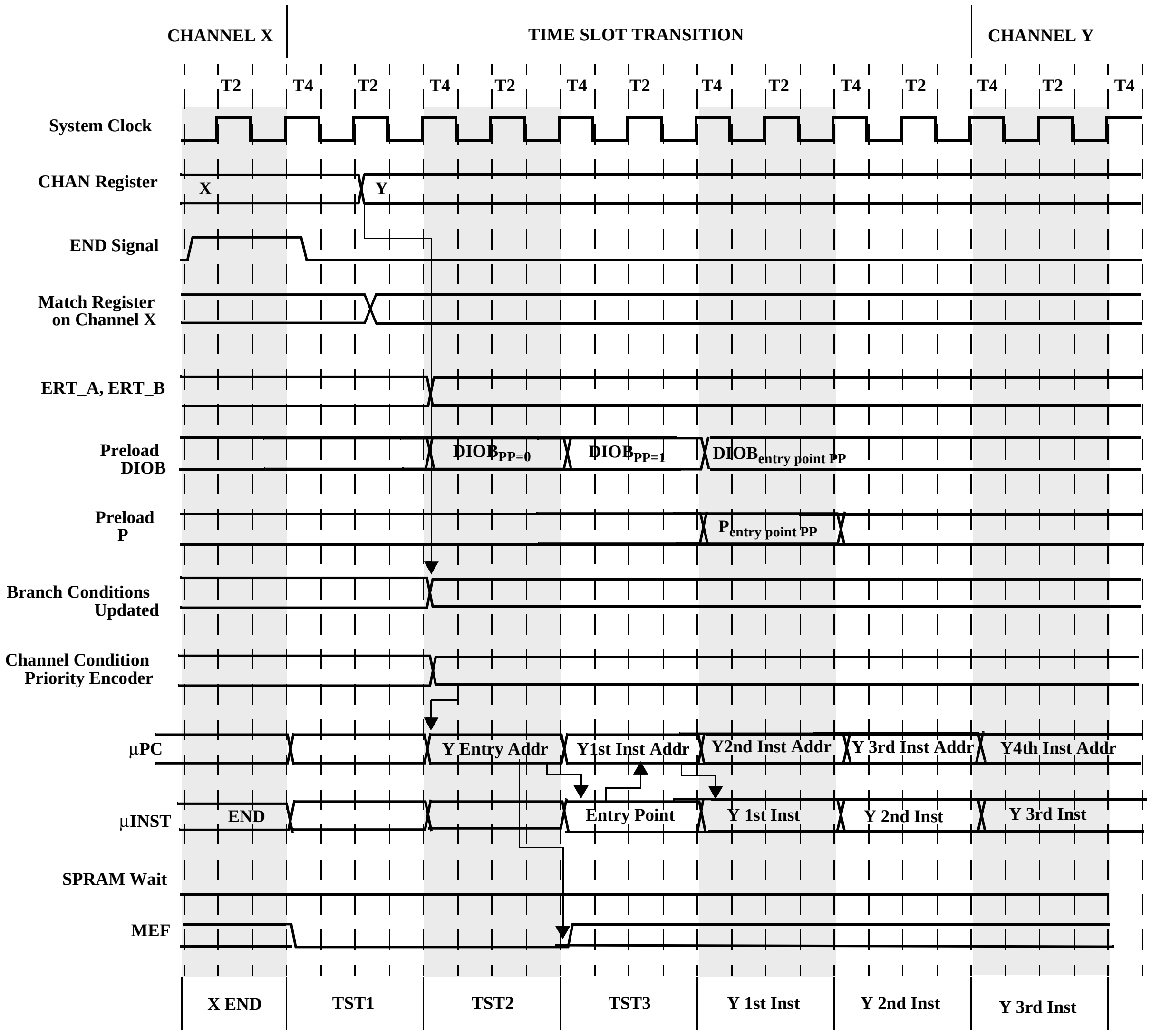


Figure 7-4. TST Timing, No Wait-states

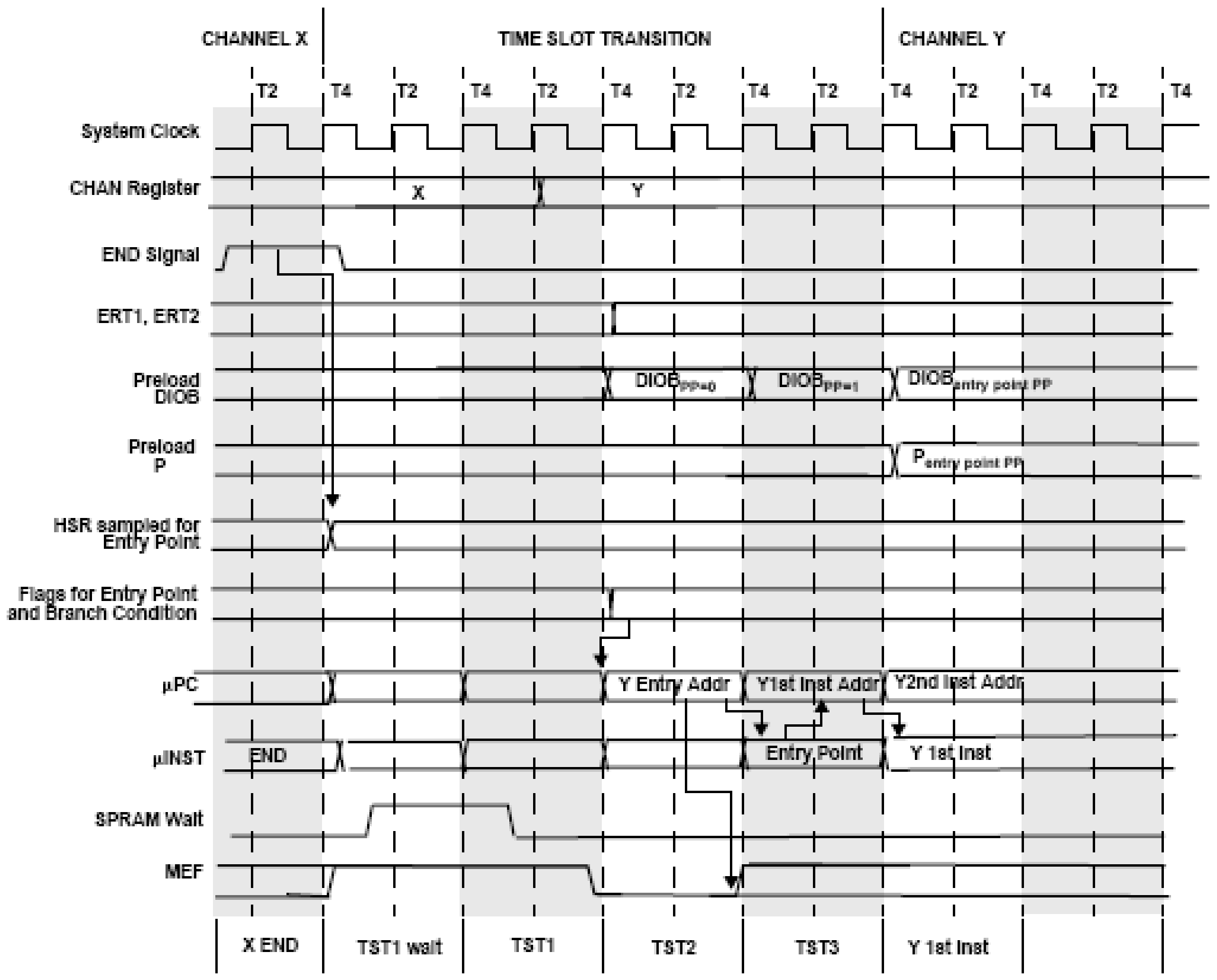


Figure 7-5. TST Timing, 1 Wait-State

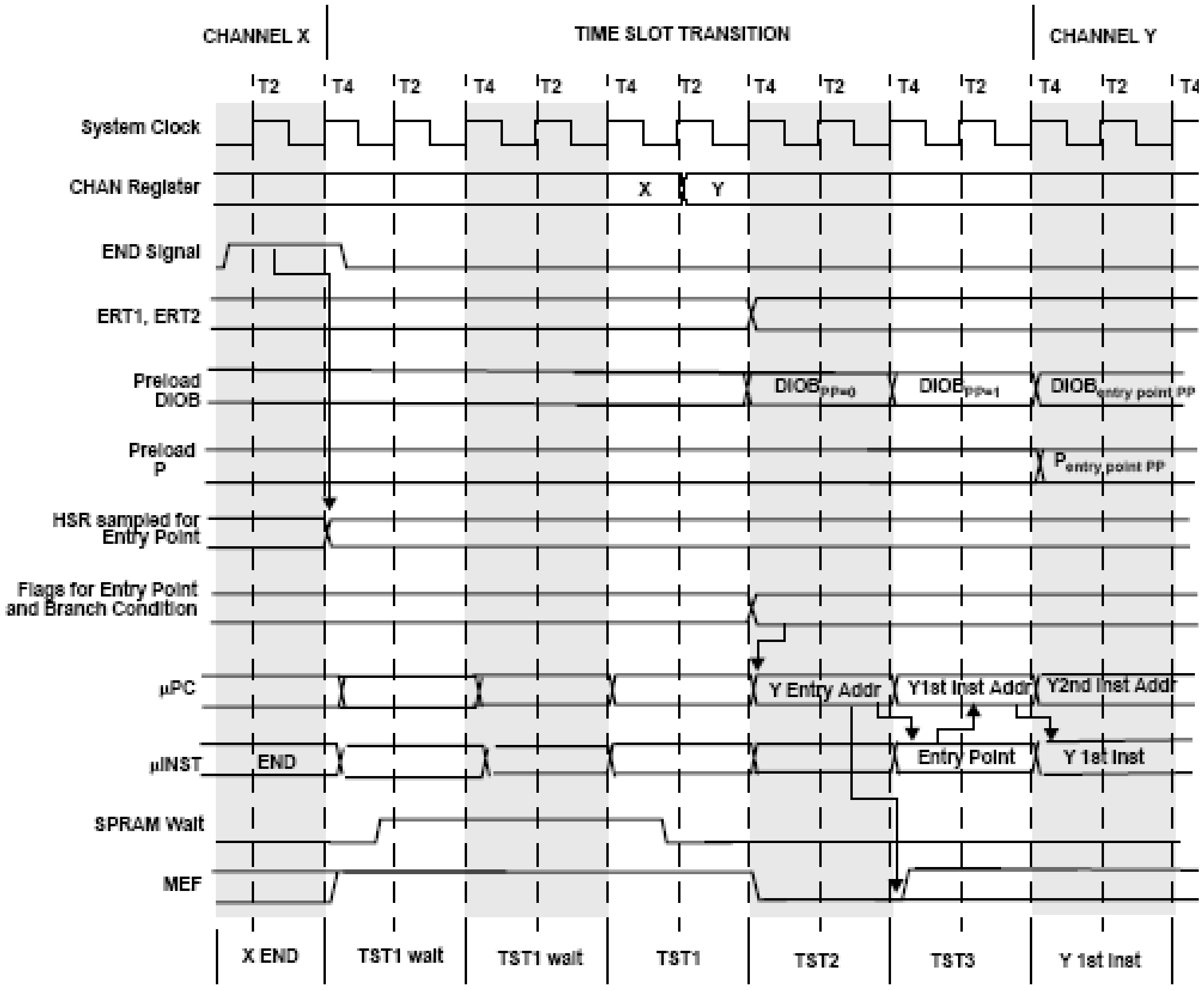


Figure 7-6. TST Timing, 2 Wait-states

For more information on channel-specific registers and flags, refer to Section 5.5, “Enhanced Channels.” For more information on P, ERT_A/B and DIOB registers refer to Section 5.8.1, “Registers.”

Chapter 8
Microengine

8.1 Introduction

Each eTPU engine has a microengine that fetches, decodes and executes microinstructions. The microengine only works when there are service requests to be attended, otherwise it turns to idle state, waiting for the next service request from the hardware scheduler, see Section 5.3, “Scheduler.”

Microcode is stored in shared code memory (SCM) which is 32 bits wide. The microengine uses a Harvard architecture to access SPRAM and code memory on different buses, so that code and data can be accessed at the same time.

The eTPU’s functionality is only possible with the microengine. The microengine allows the eTPU to have high flexibility since any desired action for a channel’s event can be implemented; however, that flexibility has a cost due to channel service latency. Latency is increased when channels from the same eTPU engine contend for microengine service. Figure 8-1 shows a block diagram of microengine architecture.

Summary of eTPU Microengine features:

- P, DIOB, A, B, C, D, SR, RAR, LINK, CHAN, MACL, MACH, ERT_A, ERT_B, TCR1, TCR2, TPR, and TRR registers are accessible by microcode.
- 24-bit ALU and Post-ALU shifter performs basic arithmetic and logical operations described in Section 8.3, “ALU and Post-ALU Shifter.”
- MDU (MAC/Divide Unit) performs integer MAC, multiply and divide operations.
- Fixed microinstruction Size of 32bits.
- Fixed-length instruction execution (2 system clocks)
- Superscalar operation

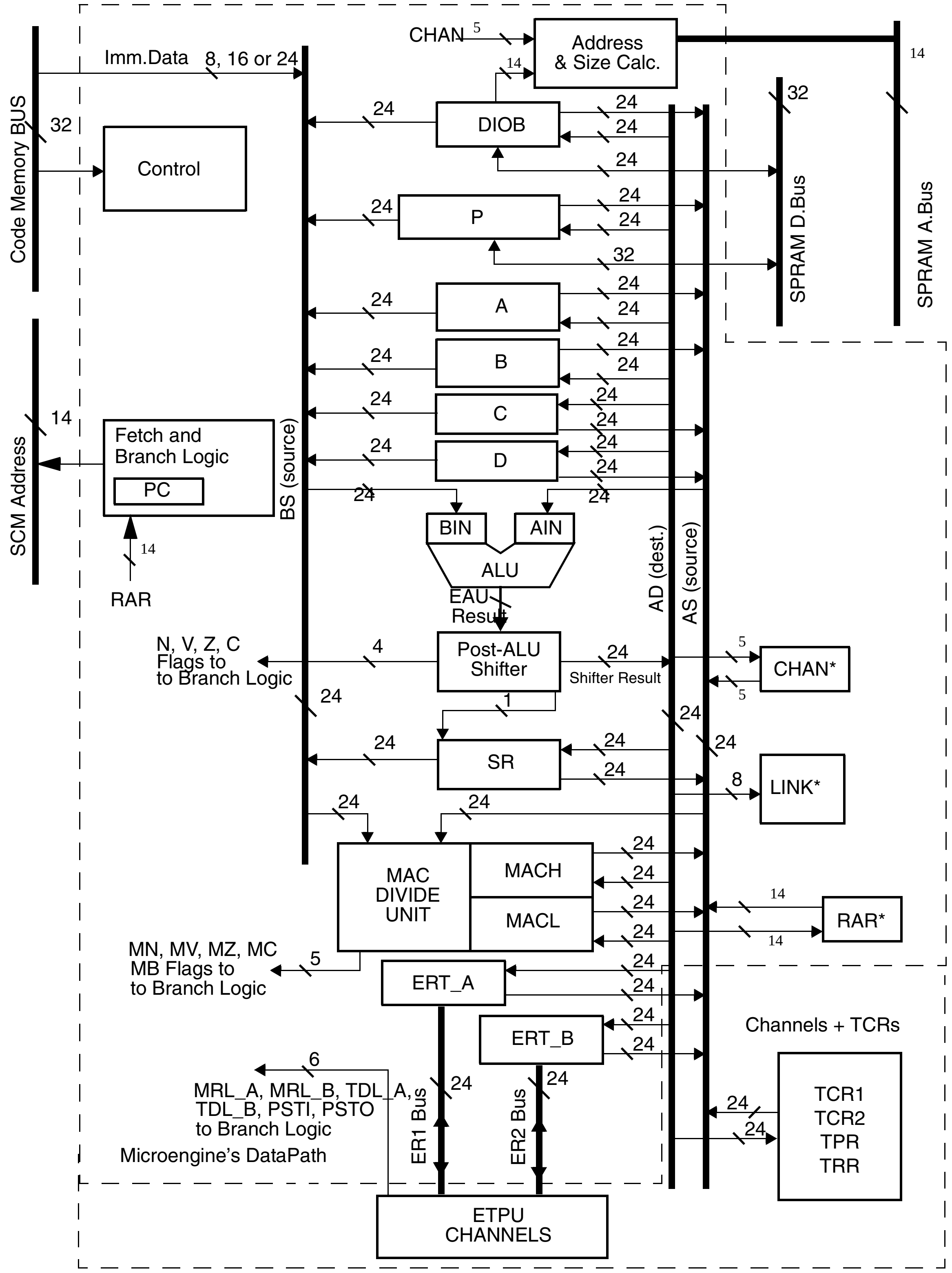


Figure 8-1. Microengine Block Diagram

8.2 Registers

eTPU microengine accesses a total of 18 registers. Sixteen of them are special purpose (only A, B, C, and D registers are for general use). Special purpose registers except CHAN and LINK can also be used as general use if the operations which use their contents are not performed. The following register descriptions are intended to just introduce their functionalities and not to provide detailed explanations of them since they will be described in Section 5.9, “Microinstruction Set.” Registers less than 24 bits in size are right-justified.

8.2.1 Preload Register (P)

The P register is the only 32-bit wide register in the eTPU. It can be used as source or destination for arithmetic/logical operations, and SPRAM read/write operations. The P register can be accessed in ALU operations in byte, half-word and word format. For P source/destination possibilities in ALU/MDU microoperations, see Section 5.9.2.2, “Selecting Sources and Destination.”

When P is used as a SPRAM read/write operation source or destination there are only 3 possibilities of access: all 32 bits, lower 24 bits and upper 8 bits. SPRAM operations are explained in detail in Section 5.9.1, “SPRAM Microoperations.”

P is automatically loaded with one SPRAM parameter before a thread starts (parameter preload). For more information see Section 5.1.1.5, “Entry Point Format,” and Section 5.1.2, “Time Slot Transition.”

The upper 8 bits of the P register may be used as flags to indicate the application state, since these bits can be tested as branch conditions. P[31:24] is also used for a dispatch microoperation, see Section 5.9.4.3, “Dispatch Microoperation,” and bit pairs P[29:28], P[27:26], P[25:24] can be directly copied into channel flags 1 and 0 using the FLC field. Together with entry table condition encoding, this data allows for fast state resolution without code execution.

8.2.2 DIOB Register

DIOB is an abbreviation for data input/output buffer. The DIOB register is 24 bits wide and can be used as source or destination for arithmetic/logical operations as well as SPRAM data. The DIOB can only be accessed as 24 bits, both in arithmetic/logical and SPRAM read/write operations. When using the DIOB to perform an SPRAM access, only the lower 24 bits of SPRAM will be accessible (the upper 8 bits always remain unchanged).

The DIOB can also be used as SPRAM addressing register, using the DIOB contents as an absolute SPRAM address (14 bits wide). When used as an address, the DIOB can also be pre-decremented or post-incremented, see Section 5.9.1.1.3, “Indirect Addressing Mode.”

The DIOB is automatically loaded with one SPRAM parameter before the thread starts (parameter preload). For more information see Section 5.1.1.5, “Entry Point Format,” and Section 5.1.2, “Time Slot Transition.”

8.2.3 Event Register Temporary (ERT_A) and (ERT_B)

ERT_A/B registers are 24 bits wide and can be used as source or destination in arithmetic/logical operations. ERT_A/B are the only sources for a write to a channel’s match register(s), see Section 5.9.3.5, “Write Channel Match Registers.”

When a thread starts to be executed, ERT_A and ERT_B are loaded with a copy of Capture1 and Capture2 registers respectively. ERT_A/B can be used to read from Match1 and Match2 registers. In fact, ERT_A/B are the only valid destination of Match1/2 read operation, see Section 5.9.2.2.2, “Special T4ABS Source Operation: Read Match Registers.”

ERT_A and ERT_B also receive a copy of Capture1 and Capture2 registers when CHAN register is written, see Section 8.2.8, “CHAN Register.” For more information about capture and match registers see Section 5.5.1.1.1, “Match1 and Match2 Registers,” and Section 5.5.1.1.2, “Capture1 and Capture2 Registers.”

8.2.4 Shift Register (SR)

SR is a 24-bit wide register that can be used as source and destination register for arithmetic/logical operations. SR is capable of a shift right by 1 bit operation. While shifting right, SR may receive in bit 23 the lost bit from a shift-right operation in post-ALU shifter, Section 8.3, “ALU and Post-ALU Shifter,” allowing SR to be used to perform 48-bit shift right, see Section 5.9.2.7, “Shift Operations.”

8.2.5 Multiply Accumulate High/Low Register (MACH) and (MACL)

Both MACH and MACL are 24-bit registers, part of MAC/divide unit, see Section 8.4, “MAC and Divide Unit (MDU).” They can be used as source and destination in most arithmetic/logic operations. When multiply or divide operations are used (multiply-accumulate included), MACH and MACL serve a special purpose and some restrictions apply, see Section 8.4, “MAC and Divide Unit (MDU),” for more information.

8.2.6 LINK Register

LINK Register is an 8-bit register and can be used only as destination in arithmetic operations. When LINK register is written, it issues a service request for the channel number and eTPU engine equal to the number written in LINK register, see Section 5.1, “Functions and Threads,” and Section 5.5.5, “Channel Link,” for information about link service request.

8.2.7 Return Address Register (RAR)

RAR is a 14-bit register and can be used as source and destination in arithmetic operations. RAR also receives the contents of the PC register when a subroutine call is executed. The contents of RAR register are loaded into PC when a return from subroutine is executed. RAR is loaded with value 0x3FFF during TST. For more information about subroutine call and return see Section 5.9.4.2, “Branch Operations,” and Section 5.9.4.4, “Return From Subroutine,” respectively.

8.2.8 CHAN Register

CHAN is a 5-bit register which can be used as source and destination in arithmetic operations. The contents of CHAN register affects the execution of many channel-related microinstructions, because its number indicates the selected channel. CHAN register must not be used to store temporary values in arithmetic operations. For more details, refer to Section 5.5.1.3.1, “Channel Selection Register (CHAN).”

8.2.9 Counter Registers: TCR1, TCR2, TPR, and TRR

All counter registers, except for TPR, are 24 bits wide. TPR is a 16-bit register. They can be read or written in arithmetic/logical operations, and serve a special-purpose when used for time base and angle mode operations. For more information about those registers see Section 5.6, “Time Bases,”and Section 5.7, “eTPU Angle Counter (EAC).”

8.2.10 General Purpose Registers: A, B C and D

A, B C and D are 24-bit general purpose registers, which can be used to store intermediate values and don’t have other specific uses with any eTPU feature.

8.3 ALU and Post-ALU Shifter

The ALU executes 24-bit arithmetic and logical operations. The ALU’s output goes directly to a 1-bit shifter, called the post-ALU shifter, so it is possible, for example, to add and shift using only one microinstruction.

All possible ALU operations can be performed with instruction formats that have the ALUOP field. These operations include add/subtract using C (carry) flag as ALU’s carry-in, bitwise and/or/not/xor, and shift/rotate of 2, 4, 8 and 16 bits. See Section 5.9.2.13, “ALU/MDU Operation Selection.” In some microinstruction formats, it is not possible to specify the operation executed by ALU. In these cases the ALU will always perform an addition operation.

Subtraction, inversion, increment and decrement can be performed by combinations of source inversion and setting ALU’s carry-in to 1.

The ALU always performs 24-bit operations on its inputs (A-source and B-source) and outputs a 24-bit result. 8- and 16-bit inputs are zero padded to 24 bits; the A-source sign can be extended from 8- or 16-bit inputs with the microinstruction field AS/CE. Likewise, a 24-bit ALU output is always truncated to the destination register size.

A-source and B-source can be selected from any of the registers except LINK, which is write-only, besides other values.

8.3.1 ALU Flags

Four flags (carry, negative, overflow, zero) described below, are related to ALU and post-ALU shift operations. Operation size and shifting affects the flags generation logic. Operation size determines the result boundary to be used for flags generation. Operation size is determined by size of sources and destination. For more information about flag generation, see Section 5.9.2.3, “Flags Sampling Control.” ALU flags can be used as branch condition, see Section 5.9.4.2.3, “Conditional/Unconditional Branch,” or Section 5.9.2.10, “Conditional ALU/MDU Operation Execution.”

Field CCS/CCSV in microinstructions can force no update of all flags. Not all flags are updated in all ALU operations: overflow is updated only on addition and absolute value operations, carry is updated in most ALU operations, and both zero and negative are updated in all ALU operations.

NOTE

Operation size can be smaller than destination register. For example: 0xFFFF + 0x0001 (both 16-bit sources) stores 0x10000 in a 24-bit register and sets zero and carry flags because operation size is 16 bits.

8.3.1.1 Carry Flag (C)

In an unsigned addition without shifting, the carry flag is the ALU carry from bit 7 to 8, 15 to 16, or 23 to 24 on 8-, 16- and 24-bit operation sizes respectively. In an unsigned subtraction without shifting, carry flag represents the sign of ALU’s result considering operation size (carry flag equal to 0 indicates a negative result).

See Section 5.9.2.7, “Shift Operations,” for more information regarding post-ALU shift operations.

The carry flag definition is operation-dependent. For the definitions of other flags, see sections below.

8.3.1.2 Negative Flag (N)

Negative flag indicates the sign of result based on the operation size, regardless of the operation performed, as shown in Table 8-1.

Table 8-1. Negative (N) Flag Behavior

Operation Size	Value
8 bits	N = result[7]
16 bits	N = result[15]
24 bits	N = result[23]

NOTE

The N flag may not reflect the sign of the value actually written into the destination register, if it does not have the same size of the operation, see Section 5.9.2.3, "Flags Sampling Control." This is always the case for registers RAR (14 bits) and CHAN (5 bits).

8.3.1.3 Overflow (V)

Overflow is updated only on addition (with or without carry) and absolute value operations. In signed operations, the overflow flag indicates that the result of arithmetic operation (add or subtraction) can not be represented by a word of the size of the operation. The overflow (V) flag behavior for addition is defined in Table 8-2. V Flag is calculated using ALU adder output (i.e., it is not affected by 1-bit shift/rotate operations).

Table 8-2. Overflow Flag on Addition[1] (V)

Op. Size	Value
8 bits	(AS[7] & BS[7] & !alu_adder_output[7]) | (!AS[7] & !BS[7] & alu_adder_output[7])
16 bits	(AS[15] & BS[15] & !alu_adder_output[15]) | (!AS[15] & !BS[15] & alu_adder_output[15])
24 bits	(AS[23] & BS[23] & !alu_adder_output[23]) | (!AS[23] & !BS[23] & alu_adder_output[23])

1. for V-flag definition on the absolute operation, see Section 8.3.8, "Absolute Value Operation."

8.3.1.4 Zero Flag (Z)

The zero flag set to 1 indicates that the result written in the destination register is zero, regardless of the operation performed. The Z flag is operation size dependent, as shown in Table 8-3.

Table 8-3. Zero Flag (Z)

Operation Size	Value
8 bits	Z = (result[7:0] == 0x00)
16 bits	Z = (result[15:0] == 0x0000)
24 bits	Z = (result[23:0] == 0x000000)

8.3.2 ALU ADD Operation with and without Shifting

The ADD operation is selected by ALUOP or ALUOPI fields when these fields are present in the microinstruction being executed and is the default ALU operation when the ALUOP or ALUOPI fields are not present. Optionally, the result can be shifted or rotated by 1 bit, which is selected by SHF, ALUOP or ALUOPI fields. See Section 5.9, “Microinstruction Set,” for more details. Table 8-4 describes how CIN and BINV fields change ADD operation behavior.

NOTE

ALU operations only occur on microinstruction formats where a destination field is found (T2ABD/T2D).

Table 8-4. Types of ADD Operations

BINV	CIN	Operation (adder output)
1	1	AS + BS
1	0	AS + BS + 1
0	0	AS - BS
0	1	AS - BS - 1

The ALU adder output can be 1-bit shifted or 1-bit rotated right as follows:

Shift right:

```
result[23:0] = adder_output[24:1]
```

Shift left:

```
result[23:1] = adder_output[22:0]

result[0]    = 0
```

Rotate right:

```
case(opsize)

8-bit:

     result[6:0]   = adder_output[7:1]

     result[7]     = adder_output[0]
```

```
      result[23:8]  = adder_output[23:8]
16-bit:
      result[14:0]  = adder_output[15:1]
      result[15]    = adder_output[0]
      result[23:16] = adder_output[23:16]
24-bit:
      result[22:0]  = adder_output[23:1]
      result[23]    = adder_output[0]
```

Table 8-5 describes carry flag behavior.

Table 8-5. Carry Flag Update on ADD Operation

BINV	Op. Size	shift/rotate	Value
1	8 bits	none	adder carry from bit 7 to bit 8
1	16 bits	none	adder carry from bit 15 to bit 16
1	24 bits	none	alu_adder_output[24]
0	8 bits	none	!adder carry from bit 7 to bit 8
0	16 bits	none	!adder carry from bit 15 to bit 16
0	24 bits	none	!alu_adder_output[24]
x	8 bits	shift left	alu_adder_output[7]
x	16 bits	shift left	alu_adder_output[15]
x	24 bits	shift left	alu_adder_output[23]
x	x	shift right	alu_adder_output[0]
x	8 bits	rotate right	adder carry from bit 7 to bit 8
x	16 bits	rotate right	adder carry from bit 15 to bit 16
x	24 bits	rotate right	alu_adder_output[24]

Flags N (negative) and Z (zero) on shift are updated according to the result after shift. Flag V (overflow) with post-alu shift is updated according to the ADD operation only, the post-alu shift does not affect the value of the V flag.

8.3.3 ADC Operation

ADC operation is selected by the ALUOP field. The CIN field is ignored when this operation is selected. Table 8-6 describes how BINV change ADC operation behavior.

Table 8-6. Types of ADC operations

BINV	CIN	Operation
1	x	AS + BS + C flag
0	x	AS - BS - C flag

The ALU Flags behave exactly the same way as for ADD operation without shift/rotate.

8.3.4 Bitwise Operations

Bitwise AND, OR and XOR are selected by ALUOP field. The CIN field is ignored for these operations and BINV field inverts (bitwise NOT) BS. C and V Flags are never updated on these operations. Table 8-7 Describes AND, OR and XOR bitwise operations.

Table 8-7. Types of Bitwise Operations

ALUOP	BINV	Operation[1]
10000	1	AS | BS
10000	0	AS | (~BS)
10001	1	AS ^ BS
10001	0	AS ^ (~BS)
10010	1	AS & BS
10010	0	AS & (~BS)

1. The logical operations are: OR represented by '|', AND represented by '&', XOR represented by '^', and NOT represented by '~' (tilde)

8.3.5 Set Bit/Clear Bit Operations

These operations set or clear the AS bit determined by BS[4:0]. If the bit number resolves to a value greater than 23, no bit is set or cleared (i.e., result is equal to AS). On these operations CIN field is ignored and BINV field inverts (bitwise NOT) BS. C and V flags are never updated for set/clear bit operations. These operations override B-Source size to 8 bits, i.e. the size of BS is considered to be 8 bits regardless of whether BS is 8, 16, or 24 bits (BS is not truncated).

Set bit (BINV = 1):

```
result = AS | (1 << BS[4:0])
```

Clear bit (BINV = 1):

```
result = AS & ~(1 << BS[4:0])
```

Set bit (BINV = 0):

```
result = AS | (1 << (31 - BS[4:0]))
```

Clear bit (BINV = 0):

```
result = AS & ~(1 << (31 - BS[4:0]))
```

8.3.6 Exchange Bit

Exchange the AS bit determined by BS[4:0] with the C flag value. If the bit number resolves to a value greater than 23, no exchange is performed (i.e., the AS and C flag values are not updated). This operation overrides the BS size to 8 bits, i.e. the size of BS is considered to be 8 bits regardless of whether BS is 8, 16, or 24 bits (BS is not truncated). On the exchange bit operation, CIN field is ignored and BINV field inverts (bitwise NOT) BS. V flag is never updated on exchange bit operation.

Exchange bit (BINV=1):

```
if BS[4:0] <= 23

begin

     temp_C_flag = AS[BS[4:0]]


     if C_flag == 1

         result = AS | (1 << BS[4:0])

     else

             result = AS & ~(1 << BS[4:0])


     C_flag = temp_C_flag

end
```

Exchange Bit (BINV = 0):

```
if (31 - BS[4:0]) <= 23

begin

         temp_C_flag = AS[31 - BS[4:0]]if C_flag == 1result = AS | (1 << (31 -
         BS[4:0]))

     else

result = AS & ~(1 << (31 - BS[4:0]))C_flag = temp_C_flag

end
```

8.3.7 Multibit Shift/Rotate Operations

These operations shift or rotate AS by 2, 4, 8 or 16 bits. Size of shift/rotate is determined by BS[1:0]. Table 8-8 describes the number of shifted/rotated bits depending on BS[1:0] value.

Table 8-8. Number of Shifted/Rotated Bits for Each BS[1:0] Value

BS[1:0]	Bits Shifted/Rotated
0	2
1	4
2	8
3	16

Shift right is a logical operation (i.e., zeros are inserted on left). Both multibit shift and rotate operations override the size of B-Source (BS) to 8 bits, i.e. the size of BS is considered to be 8 bits regardless of whether BS is 8, 16, or 24 bits (BS is not truncated).

The V flag is never updated for multibit shift or rotate operations. Carry flag behavior is described on Table 8-9.

Table 8-9. Carry Flag Value on Multibit Shift/Rotate Operations

ALUOP	BS[1:0]	C Flag Value
11001 (shift left)	0	AS[22]
11001 (shift left)	1	AS[20]
11001 (shift left)	2	AS[16]
11001 (shift left)	3	AS[8]
11010 (shift right)	0	AS[1]
11010 (shift right)	1	AS[3]
11010 (shift right)	2	AS[7]
11010 (shift right)	3	AS[15]
11011 (rotate right)	0	AS[2]
11011 (rotate right)	1	AS[4]
11011 (rotate right)	2	AS[8]
11011 (rotate right)	3	AS[16]

8.3.8 Absolute Value Operation

Absolute value operation is selected by the ALUOP field. For this operation, AS is interpreted as a signed number and its absolute value is the result. V and N flags are updated with the result signal determined by the operation size. After size override and sign extension, if any, see Section 5.9.2.11, “A-Source Size Override,” bit 23 of A-source is used to check the operand signal and is copied to C-flag regardless of the data size from A-source register. Note that if the data in AS is 8 bits or 16 bits, its sign is copied to C only if sign-extension is performed. This operation is independent of B-source. Instruction fields T4BBS, BINV and CINV are ignored in this operation.

Table 8-10. ALU Flags in Absolute Value operation

Operation Size	V, N[1]	C	Z
8	alu_output[7]	AS[23]	alu_output[7:0] == 0
16	alu_output[15]		alu_output[15:0] == 0
24	alu_output[23]		alu_output[23:0] == 0

1. V, N can be 1 on 8- and 16-bit Absolute Value, because the operand sign is always taken from bit 23. V, N can also be 1 in 23-bit Absolute Value (or 8-bit and 16-bit with sign extension), if the operand is 0x800000 (0x80, 0x8000).

8.4 MAC and Divide Unit (MDU)

The MDU is an autonomous resource in the microengine which can carry out sequential multiply, multiply-accumulate, fractional multiplication and divide operations, selected through the microinstruction fields ALUOP or ALUOPI. The unit supports signed and unsigned multiply and fractional multiplication of any combination of 8, 16 or 24 bit operands, see Note: , and also signed and unsigned 24-bit multiply-accumulate. The divide operation is unsigned, and both operands are always 24-bit wide.

Depending on the size of operands and the type of operation, the MDU can take more than one microcycle to execute the operation, but the microengine continues to execute microinstructions in parallel. When the microcode issues an END command, any operation executing in the MDU terminates immediately and is left incomplete. When selecting an operation that uses the MDU, the result is always placed in MACH and MACL registers, and the register selected as the destination by the microcode is not written, Section 5.9.2.2, “Selecting Sources and Destination.” During calculations, MACH and MACL hold the temporary values and should not be written, otherwise the result is unpredictable. A new MDU operation should almost never be started when one is in progress, see Section 8.4.1, “Multiply and Multiply-Accumulate Operation Length.” The result of starting a new operation will be unpredictable for both the operation in progress as well as the new one.

MDU Operations update the MDU’s own set of 5 flags, described in Section 8.4.10, “MDU Flags.” MDU operations never update C, N, V and Z flags. CIN and BINV microinstruction fields affect MDU operations according to Table 8-11.

Table 8-11. CIN and BINV with MDU Operations

B-source Operand	BINV	CIN	Operation Performed
signed	1	1	AS signed_mdu_op BS
	0	0	AS signed_mdu_op (-BS)
	1	0	Reserved
	0	1	Reserved
unsigned[1]	1	1	AS unsigned_mdu_op BS
	1	0	AS unsigned_mdu_op (BS+1)
	0	x	Reserved

1. includes the B-source (unsigned) in fmults (signed) operations.

8.4.1 Multiply and Multiply-Accumulate Operation Length

The MDU needs 2 sources, A source and B source, to perform an operation (A-source and B-source are selected the same way as in ALU operations). The time needed to perform a multiply or multiply-accumulate is:

- B-source (BS) size = 8 bits: 2 microcycles (one start-MDU plus one execution microcycle)
- BS size = 16 bits: 3 microcycles (one start-MDU plus two execution microcycle)
- BS size = 24 bits: 4 microcycles (one start-MDU plus three execution microcycle)

An internal pipeline in the MDU unit allows multiply or multiply accumulate operations to start one microinstruction before the last one has been completed (e.g., one can start one multiply with 8-bit B source in every microinstruction). However, by doing that it is not possible to read the result in MACH and MACL, so this is intended to be used in a multiply-accumulate sequence.

Multiply-accumulate operations are similar to multiply operations, except that the contents of MACH and MACL registers are added to the multiplication result.

When multiply or multiply accumulate operations finish, MACL and MACH hold the least and the most significant 24-bit words, respectively.

8.4.2 Divide Operation Length

The divide operation is always unsigned. It takes 13 microcycles to complete the calculation, meaning that after the start divide microinstruction, one has to wait for 11 microcycles and then read the result and the remainder in MACH and MACL registers. During the 11 divide execution microcycles, microengine can execute microinstructions unrelated to the MDU.

8.4.3 Signed Multiplication (mults)

MDU signed multiplication is defined as follows:

```
(signed) MACH,MACL = (signed) AS * (signed) BS
```

MC and MV flags are reset. MZ is set if result is 0; MZ resets otherwise. MN is set if result is negative.

8.4.4 Unsigned Multiplication (multu)

MDU unsigned multiplication is defined as follows:

```
(unsigned) MACH|MACL = (unsigned) AS * (unsigned) BS
```

MC and MV flags are reset. MZ is set if result is 0, and MZ resets otherwise. MN is a copy of the most significant bit of result.

8.4.5 Signed Multiply-Accumulate (macs)

MDU signed multiply-accumulate is defined as follows:

```
(signed/unsigned) {MACH,MACL} += (signed) AS * (signed) BS
```

MC is not altered.

MV is set if result can not be represented by a 48-bit signed number. **macs** never resets the MV flag: it is left as is if no overflow occurs, or set it otherwise. This allows checking the overflow flag only once at the end of a series of multiply-accumulate operations in a scalar product calculation.

```
if ({MACH,MACL} += AS * BS < $-2^{47}$ || {MACH,MACL} += AS * BS > $2^{47}$ - 1)
      MV = 1
```

MZ is set if result is 0. MZ resets otherwise. MN is a copy of the most significant bit of result.

Note that only 24-bit multiply-accumulate is available.

8.4.6 Unsigned Multiply-Accumulate (macu)

MDU unsigned multiply-accumulate is defined as follows:

```
(signed/unsigned) {MACH,MACL} += (unsigned) AS * (unsigned) BS
```

MC is set if the result cannot be represented by a 48-bit unsigned non-negative number. MACU never resets MC flag; the MC flag is left as is if no carry occurs, or set otherwise.

This allows checking the carry flag only once at the end of a series of multiply-accumulate operations in a scalar product calculation.

```
if ({MACH,MACL} += AS * BS < 0 || {MACH,MACL} += AS * BS > 2^48 - 1)
      MC = 1
```

MV is not altered.

MZ is set if result is 0; MZ resets otherwise. MN is a copy of the most significant bit of result.

Note that only 24-bit multiply-accumulate is available.

8.4.7 Signed Fractional Multiplication (fmults)

MDU signed fractional multiplication takes the B-source as an unsigned 8- or 16-bit fraction between 0 and $(2^8 - 1)/2^8$ (inclusive) for the 8-bit operation, or between 0 and $(2^{16}-1)/2^{16}$ (inclusive) for the 16-bit operation. Only A-source is taken as a signed number. The value of B-source is considered the unsigned numerator of a fraction with denominator 2^8 or 2^{16} for the 8- and 16-bit operations, respectively.

The integer part of the result is stored in MACH, and the fractional part in MACL. The result is signed, so that the concatenation of MACH and MACL forms a 48-bit fixed point number with a 24-bit mantissa, both for 8- and 16-bit operations. To calculate the unsigned numerator of the fractional part (with denominator 2^{24}) of the result, one must take the absolute value of MACL considering the sign of the result, (both MACH and MACL, not MACL alone), i.e.: if flag MN=1, invert MACL and add 1.

MDU flags are updated in the same way as in the signed multiplication opperation.

NOTE

There is no distinct selection of 24-bit fractional multiplication, for it works exactly as a 24-bit ordinary multiplication.

8.4.8 Unsigned Fractional Multiplication (fmultu)

MDU unsigned fractional multiplication takes both A-source and B-source as unsigned operands. B-source is taken as an 8- or 16-bit fraction between 0 and $(2^8 - 1)/2^8$ (inclusive) for the 8-bit operation, or between 0 and $(2^{16}-1)/2^{16}$ (inclusive) for the 16-bit operation. The value of B-source is considered the numerator of a fraction with denominator 2^8 or 2^{16} for the 8- and 16-bit operations, respectively.

The integer part of the result is stored in MACH, and the fractional part in MACL. The fractional part in MACL is the numerator of a fraction with denominator 2^{24}. The concatenation of MACH and MACL form a 48-bit fixed point number with a 24-bit mantissa, both for 8- and 16-bit operations.

MDU flags are updated in the same way as in the unsigned multiplication.

NOTE

There is no distinct selection of 24-bit fractional multiplication, for it works exactly as a 24-bit ordinary multiplication.

8.4.9 Unsigned Divide (div)

At the end of a divide operation MACL holds the result of the division, while MACH holds the remainder. If a divide by 0 is executed, MACL holds the maximum unsigned number (0xFFFFFF) as result and flag MV is set to indicate division by 0 (otherwise reset). The contents of MACH are not defined for a divide-by-0.

MC flag is always reset.

MZ flag is set if MACL equals 0, and reset otherwise.

MN receives a copy of MACH bit 23 (the msb from the remainder).

Note that signed division is not available.

8.4.10 MDU Flags

MDU has its own flags to indicate the result and status of an MDU operation. They are: MC, MZ, MV, MN and MB.

8.4.10.1 MDU Negative Flag (MN)

MN flag is always a copy of the most significant bit of the result, for both signed and unsigned operations.

8.4.10.2 MDU Carry Flag (MC)

MDU carry flag indicates if the result cannot be represented by a 48-bit number, in signed and unsigned multiply accumulates. It is reset in the other operations.

8.4.10.3 MDU Zero Flag (MZ)

In multiply and multiply-accumulate operations, the MDU zero flag is asserted if MACH and MACL are equal to zero at the end of an operation. In divide operations, zero flag is asserted if MACL (result) is equal to 0.

8.4.10.4 MDU Overflow Flag (MV)

In multiply operations, the MV flag is negated and remains negated at the end because the result of a 24 x 24-bit multiplication can always fit in a 48-bit result (MACH and MACL concatenated). In a multiply-accumulate operation, MV is asserted if the result size is wider than 48 bits. The MV flag works in both signed and unsigned operations.

In divide operations MV is only asserted if a divide-by-zero operation was executed.

8.4.10.5 MDU Busy Flag (MB)

When asserted, MB indicates that MDU is calculating, otherwise it indicates that MDU is idle. MB tests "true" at the next microinstruction after the MDU start operation. MB tests "false" at the last microcycle of any MDU operation execution.

8.5 Branch Conditions

The microengine allows conditional branching. There are five sets of flags which can be tested in a conditional branch: ALU flags, MDU flags, P flags, channel flags, and semaphore flag (flag SMLCK).

When a thread starts to be executed, the values in MDU and ALU flags are not initialized. ALU flags are described in Section 8.3.1, "ALU Flags," MDU flags are described in Section 8.4.10, "MDU Flags." MDU and ALU flags are updated during execution of microinstructions.

P flags are actually the upper byte of P register, which may be utilized as user defined flags, see Section 8.2.1, "Preload Register (P)."

Channel Flags MRL_A, MRL_B, TDL_A, TDL_B, PSS, PSTI and PSTO, see Note: Channel "state resolution" flags Flag0 and Flag1 cannot be tested by microcode., are obtained from the selected channel (value in CHAN register), while channel flags, LSR, FM[0] and FM[1] are selected by the serviced channel, regardless of the CHAN value.

NOTE

Channel "state resolution" flags Flag0 and Flag1 cannot be tested by microcode.

NOTE

The channel being serviced does not change during execution of a thread (a thread is atomic), and it is the channel that requested a service (initial value of CHAN register when a thread starts).

Flags TDL_A/B, MRL_A/B, LSR, FM[1:0] and PSS, are sampled at the beginning of a thread. The PSS flag value cannot change during the thread's execution until after the

CHAN register is written. When a write in CHAN register is performed, all flags except LSR and FM[1:0] are updated according to the channel specified by CHAN value. Flags MRL_A/B and TDL_A/B are reset when their respective channel latches are cleared by microcode

Table 8-12. Channel Flags as Branch Condition

Flag	Description	Service or Selected Channel
MRL_A	Match1 Recognition Latch	These flags reflect the selected channel (CHAN) see Section 5.5.2.1, "Match Recognition Latches (MRL1/2)," and 5.5.3.1 for more information.
MRL_B	Match2 Recognition Latch	
TDL_A	Transition1 Detection Latch	
TDL_B	Transition2 Detection Latch	
LSR	Link Service Request	Reflects the serviced channel.
PSS	Sampled Input Pin State	Reflects the selected channel (CHAN). Does not change if CHAN is not changed, see Section 5.5.1.2, "Pin Control Registers."
PSTI	Current Input Pin State.	Reflects the selected channel (CHAN). May change any time.
PSTO	Current Output Pin State	Reflects the selected channel (CHAN). May change any time.
FM[1:0]	Function Mode Bits	Reflects the function mode for serviced channel, Section 4.6.2, "eTPU Channel x Status Control Register (ETPUCxSCR)."

Semaphore condition SMLCK indicates if a semaphore is locked for the engine. The branch resolves as false before a lock attempt is made. For each trial, the SMLCK flag is updated. The SMLCK value set in one thread is not meaningful to the other. After a free, the SMLCK condition tests as false until a new lock attempt on the same thread.

Chapter 9 Microinstruction Set

9.1 Introduction

Each microinstruction can execute up to 3 microoperations in parallel. Microinstructions are grouped into formats, and there are four types of microoperations:

- ALU/MDU operations
- SPRAM operations
- Channel configuration/control operations
- Flow control operations

Each microinstruction format is defined by a set of microinstruction fields, which determine the operations, each belonging to one of the four groups above (there may be several fields belonging to one group in a microinstruction). Complete microinstruction formats are shown in Section 9.8, "Microinstruction Formats."

Parallelism conflicts may arise when two operations are executed in the same microinstruction. These situations are explained in Section 9.7, "Microinstruction Parallelism Issues."

9.2 SPRAM Microoperations

The access to SPRAM is made by providing an address and a register to perform a data transfer, except for semaphore operations, which are also classified in the SPRAM group. Only P and DIOB registers can exchange data with SPRAM. The microengine always addresses SPRAM in 32-bit boundaries, for 8, 24, or 32 bit wide data.

The data transfer direction is determined by the field RW in all addressing modes: RW=0 selects read from SPRAM and RW=1 selects write to SPRAM.

9.2.1 SPRAM Addressing Modes

There are 3 eTPU addressing modes:

- Absolute

- Selected channel relative
- Indirect

The absolute and selected channel relative addressing modes use immediate bits to form the SPRAM physical address, which is identified in microinstruction as a field called AID. The AID field can be 3-, 7-, or 8-bit wide depending on the addressing mode.

9.2.1.1 Absolute Addressing Mode

In absolute addressing mode, the address range is 256 parameters, addressed by the field AID, which in this mode is 8 bits wide. These parameters are located in SPRAM addresses from 0 to 255.

```
physical_address = AID[7:0]*4
```

9.2.1.2 Selected Channel Relative Addressing Mode

In selected channel relative addressing mode, only the first 8 (with 3-bit AID) or 128 (with 7-bit AID) parameters of the selected channel are accessible, depending on the microinstruction format. The physical address is calculated using the channel parameter base address that is specified in field CPBA of ETPUCxCR register, see Section 4.6.1, “eTPU Channel x Configuration Register (ETPUCxCR).”. The AID field is added to channel parameter base address to compose the physical address. The equation is:

```
physical_address = selected_channel_parameter_base_address + AID[6:0]*4, or


physical_address = selected_channel_parameter_base_address + AID[2:0]*4
```

9.2.1.3 Indirect Addressing Mode

In indirect addressing mode the physical address is taken from DIOB register. Only DIOB bits 13 to 2 are relevant. Since the SPRAM word address is shifted two bits up in DIOB, its contents hold the same parameter address value used by the host. The equation is:

```
physical_address = DIOB[13:2]*4, or

physical_address = DIOB & 0x003FFC
```

Indirect addressing mode can post-increment or pre-decrement DIOB, allowing stack operations. See Section 9.2.6, “DIOB Stack Operation,” for more information.

9.2.2 SPRAM Source/Destination Registers

When performing an SPRAM operation, only DIOB or P can be used as data source or destination. P is 32-bit wide, and DIOB is 24-bit wide. Microinstruction field P/D (1 bit) is used to choose between P and DIOB as data the source or destination. When the P/D field is not available in microinstructions that support SPRAM access, the destination is P.

Table 9-1. SPRAM Source/Destination Register Selection

P/D	Meaning
0	P access
1	DIOB access

9.2.3 SPRAM Operation Size

When using DIOB register to perform SPRAM data transfers, the operation size is always 24 bits wide (lower 24 bits of SPRAM). When using P register, the operation size can be 8, 24, or 32 bits wide, which is controlled by microcode RSIZ field (2 bits). RSIZ options are shown in Table 9-2.

Table 9-2. SPRAM P Access Size

RSIZ	Meaning
00	full 32-bit access (i.e. P[31:0]=SPRAM[addr] [31:0])
01	only upper 8 bits are transferred (i.e. P[31:24] = SPRAM[addr] [31:24])
10	only lower 24 bits are transferred (i.e. P[23:0] = SPRAM[addr] [23:0])
11	RESERVED

RSIZ is not available for all microinstructions that support SPRAM access. In microinstructions where RSIZ field is not available, the default SPRAM access is 24 bits.

When performing a ZERO SPRAM write operation, see Section 9.2.5, “Zero SPRAM Operation,” RSIZ defines the size of operation regardless of the P/D field, Section 9.2.2, “SPRAM Source/Destination Registers.”

9.2.4 SPRAM Access Direction

RW field defines the direction of the access in the SPRAM. The access direction is summarized in Table 9-3.

Table 9-3. SPRAM Access Direction

R/W	Meaning
0	read SPRAM parameter into P or DIOB registers
1	write SPRAM parameter from P or DIOB registers

9.2.5 Zero SPRAM Operation

The zero SPRAM operation is controlled by microcode field ZRO (1bit). When ZRO field is 0, data written in SPRAM or in P/DIOB (SPRAM read) registers will always be 0x0. When performing a zero SPRAM write operation, the RSIZ determines the size of the write regardless of the P/D field (usually RSIZ is meaningful only for P/D = 0), which means that zero SPRAM write operation can be performed with 32, 24 or 8 bits according to SPRAM operation size. These conditions are summarized in Table 9-4.

Table 9-4. Zero SPRAM Operation

ZRO	RW	P/D	Meaning
0	0	0	Clear P register. Size is determined by RSIZ field. See Section 9.2.3, “SPRAM Operation Size.”
0	0	1	Clear DIOB (all 24 bits), independently of RSIZ
0	1	x	Clear SPRAM parameter. Size is determined by RSIZ field. See Section 9.2.3, “SPRAM Operation Size.”
1	RW	P/D	Regular SPRAM operation

NOTE

When STC field is present in a microinstruction, STC=11 will disable the zero SPRAM operation (see Table 9-5).

9.2.6 DIOB Stack Operation

SPRAM indirect addressing mode, see Section 9.2.1.3, “Indirect Addressing Mode,” is used if the STC field (2 bits) is in the microinstruction. STC controls automatic increment/decrement of DIOB register, as shown in Table 9-5, thus allowing stack operations. **Only** DIOB bits 15 to 2 are incremented and decremented, i.e.: bits 23 to 16 and 1 to 0 are **not** touched by STC pre-decrement and post-increment.

Table 9-5. DIOB Post-Increment / Pre-Decrement (STC)

STC	Meaning
00	Post-Increment of DIOB
01	Pre-Decrement of DIOB
10	No Increment/Decrement (normal access)
11	No SPRAM Access[1]

1. this disables the Zero SPRAM operation.

9.2.7 Semaphore Operations

Semaphore lock and free operations are controlled by eTPU microcode. For more information about semaphores see Section 5.1.4, “Hardware Semaphores.” There are 2 microcode fields that control semaphore operations: FL (1 bit) and SMPR (2 bits). A serviced channel sees 4 semaphores, selected by field SMPR.

Table 9-6. Semaphore Operations Fields

Field	Meaning
FL	0 = free semaphore, 1 = lock semaphore
SMPR	semaphore number selector

Only one semaphore can be locked at a time by each engine, so when freeing a semaphore it is not necessary to specify its number. Thus, when FL = 0 (free semaphore), SMPR has no effect.

NOTE

If the microcode tries to lock an engine's semaphore already locked for the same engine, the semaphore remains locked and the SMLCK branch condition resolves as true. Since the semaphore is always unlocked when the thread ends, the only reason it could be locked is if it were locked in the current thread.

9.3 ALU/MDU Operations

ALU/MDU microoperations are usually are composed of 2 sources, 1 destination and 1 operation. The operation is generally selected through fields ALUOP, ALUOPI or SHF. In formats where there is no operation selection field (ALUOP, ALUOPI or SHF), the operation performed is always addition; however, it is possible to perform subtraction, increment or decrement using fields BINV, see Section 9.3.4, "B-Source Inversion," and CIN, see Section 9.3.5, "Carry-in Control."

9.3.1 A-Source and Destination Register Set Selection

The microcode field T4ABS allows selection of a source from either one of two register sets, shown in Table 9-10. The same applies to T2ABD, used for ALU destination selection with other two register sets, as shown in Table 9-11. When available in the microinstruction format, fields ABSE and ABDE select one of the two register sets for source and destination, respectively. In formats without ABSE/ABDE, the T4BBS field determines the register sets used by T2ABD and T4ABS, as shown in Section 9.3.1.2, "Microinstructions Without Fields ABSE and ABDE."

NOTE

B-source selection is done by the T4BBS field. T4BBS also selects register sets, see Section 9.3.1.2, “Microinstructions Without Fields ABSE and ABDE.”

9.3.1.1 Microinstructions With Fields ABSE and ABDE

In microinstructions where ABSE and ABDE fields are available (1 bit each), ABSE controls register set selection for T4ABS (source) and ABDE controls register set selection for T2ABD (destination). Table 9-7 shows the meaning of values for ABSE and ABDE fields.

Table 9-7. Register Set Selection by ABSE or ABDE

ABSE or ABDE	Register Set Selected
0	second
1	first

9.3.1.2 Microinstructions Without Fields ABSE and ABDE

When ABSE and ABDE are not available in a microinstruction format, the register sets for T4ABS and T2ABD are specified by the T4BBS field, as illustrated by Table 9-8.

Table 9-8. Register Set Selection by T4BBS w/o ABSE,ABDE

T4BBS	Register Set For T2ABD	Register Set For T4ABS
0xx	first	first
100	second	second
101	second	first
110	first	second
111	first	first
none[1]	first	first

1. refers to operations with immediate data as B-source, without ABSE,ABDE.

9.3.2 Selecting Sources and Destination

All ALU/MDU operations need 2 sources (called AS and BS) and ALU operations also need 1 destination (called AD), except for some of those that use immediate data, see Section 9.3.14, “Operations With Immediate Data.” Fields T4ABS (4 bits), ABSE (1 bit), T4BBS (3 bits) select sources, while T2ABD (4 bits) and ABDE (1 bit) select the destination.When the MDU is used (multiply/divide), T2ABD destination selection is ignored and results are stored in MACH and MACL, see Section 5.8.3, “MAC and Divide Unit (MDU).” ABSE and ABDE are not available in some microinstruction formats that support ALU/MDU operations. ABSE and ABDE are always coupled in microinstruction formats that feature these fields. The existence of ABSE/ABDE fields in a microinstruction changes the meaning of T4BBS field, as shown in Table 9-9. On instructions with immediate data, T4BBs is used as B-source, see Section 9.3.14, “Operations With Immediate Data.”

All sources and destinations have a size associated with them, and these sizes are used to select the flag sample position, see Section 9.3.3, “Flags Sampling Control.” The sizes can be 8, 16 or 24 bits. Registers that are not exactly of one of these sizes are promoted to the next size up, e.g. CHAN[4:0] is an 8-bit source. See Section 9.3.3, “Flags Sampling Control,” for more information.

Some parallelism issues arise when selecting P, DIOB, ERT_A or ERT_B as destination registers, since they can be modified by other microoperations in the same microinstruction, see Section 9.7, “Microinstruction Parallelism Issues for details.”

Table 9-9. B Source Selection (T4BBS)

T4BBS	Meaning in microinstruction formats with ABSE/ABDE	Meaning in microinstruction formats without ABSE/ABDE[1]
000	BS[23:0] = P[23:0]	
001	BS[23:0] = A[23:0]	
010	BS[23:0] = SR[23:0]	
011	BS[23:0] = DIOB[23:0]	
100	reserved	BS = 0
101	reserved	BS = 0
110	reserved	BS = 0
111	BS=0, or Max const., if CIN=0 and BINV=0, see Section 9.3.6, “Generating “Max” Constant.”	

1. T4BBS also selects A-source and destination register set in this case, according to Table 9-8.”

T4ABS selects one source from 2 register sets, shown in Table 9-10. ABSE and T4BBS control which set T4ABS field uses to select the source. For more information about how to select a register set for T4ABS and T2ABD see Section 9.3.1, “A-Source and Destination Register Set Selection.” All sources are zero-filled to 24 bits, unless sign-extension is specified, see Section 9.3.11, “A-Source Size Override.”

Table 9-10. A Source Selection (T4ABS)

T4ABS	First Register Set		Second Register Set	
	Selected Register	Size	Selected Register	Size
0000	AS[7:0]=P[7:0]	8	AS[7:0]=0	8
0001	AS[7:0]=P[15:8]	8	AS[23:0]=C[23:0]	24
0010	AS[7:0]=P[31:24]	8	AS[15:0] = TPR[15:0]	16
0011	AS[23:0] = ERT_B[23:0]	24	AS[23:0] = B[23:0]	24

Table 9-10. A Source Selection (T4ABS)

T4ABS	First Register Set		Second Register Set	
	Selected Register	Size	Selected Register	Size
0100	AS[23:0] = D[23:0]	24	AS[23:0] = TRR[23:0]	24
0101	AS[15:0] = P[15:0]	16	AS[7:0] = 0, read_match, Section 9.3.2.2, “Special T4ABS Source Operation: Read Match Registers.”	8
0110	AS[15:0] = P[31:16]	16	AS[13:0] = RAR[13:0]	16
0111	AS[7:0] = P[23:16]	8	AS[23:0] = MACH[23:0]	24
1000	AS[23:0] = P[23:0]	24	AS[23:0] = MACL[23:0]	24
1001	AS[23:0] = A[23:0]	24	AS[4:0]=CHAN[4:0]	8
1010	AS[23:0] = SR[23:0]	24	AS[14:2] = CHAN_BASE, see Section 9.3.2.3, “CHAN_BASE as a Source.”	16
1011	AS[23:0] = DIOB[23:0]	24	Reserved	—
1100	AS[23:0] = TCR1[23:0]	24	Reserved	—
1101	AS[23:0] = TCR2[23:0]	24	Reserved	—
1110	AS[23:0] = ERT_A[23:0]	24	Reserved	—
1111	AS[23:0] = 0	24	Reserved	—

T2ABD selects the destination from 1 of 2 register sets, shown in Table 9-11. ABDE and T4BBS control which set T2ABD field uses to select the destination.

Table 9-11. Destination Selection (T2ABD)

T2ABD	First Register Set		Second Register Set	
	Selected Register	Size	Selected Register	Size
0000	A[23:0] = AD[23:0]	24	C[23:0] = AD[23:0]	24
0001	SR[23:0] = AD[23:0]	24	LINK[4:0] = AD[4:0]	8
0010	ERT_A[23:0] = AD[23:0][1]	24	TPR[15:0] = AD[15:0]	16
0011	ERT_B[23:0] = AD[23:0][2]	24	B[23:0] = AD[23:0]	24
0100	DIOB[23:0] = AD[23:0]	24	CHAN[4:0] = AD[4:0]	8
0101	P[15:0] = AD[15:0]	16	D[23:0] = AD[23:0]	24
0110	P[31:16] = AD[15:0]	16	RAR[12:0] = AD[12:0]	16
0111	P[23:0] = AD[23:0]	24	MACH[23:0] = AD[23:0]	24
1000	TCR1[23:0] = AD[23:0]	24	MACL[23:0] = AD[23:0]	24
1001	TCR2[23:0] = AD[23:0]	24	Reserved	—
1010	P[31:24] = AD[7:0]	8	Reserved	—
1011	P[23:16] = AD[7:0]	8	Reserved	—

Table 9-11. Destination Selection (T2ABD)

T2ABD	First Register Set		Second Register Set	
	Selected Register	Size	Selected Register	Size
1100	P[15:8] = AD[7:0]	8	Reserved	—
1101	P[7:0] = AD[7:0]	8	Reserved	—
1110	TRR[23:0] = AD[23:0]	24	Reserved	—
1111	no destination selected[3]	24	Reserved	—

1. T2ABD=0010 also writes to Match1 register of the selected channel if field ERW1=0.
2. T2ABD=0011 also writes to Match2 register of the selected channel if field ERW2=0.
3. if no destination is selected, ALU flags are updated, although the result is lost.

9.3.2.1 Max Const Generation With T4BBS=111

When T4BBS = 111, BINV = 0, and CIN = 0, the value assigned to BS will be 0x800000, and not 0x0, as would be expected. See Section 9.3.6, “Generating “Max” Constant,” for a detailed explanation.

9.3.2.2 Special T4ABS Source Operation: Read Match Registers

When T4ABS = 0101 and the source for T4ABS is selected from the second register set, the constant 0x00 is used as AS (8-bit size) and the following register transfer is performed in parallel as well: match registers of the selected channel (value in CHAN register) are copied to ERT_A/ERT_B registers, where ERT_A receives the value of Match1 register and ERT_B receives the value of Match2 register, see Section 5.2.1.1, “Event Registers (ER).” Note that ALU destination can still be chosen by T2ABD in parallel. A parallelism issue arises When ERT_A or ERT_B is selected by T2ABD, see Section 9.7.1, “ALU Operations and Read Match Registers.”

9.3.2.3 CHAN_BASE as a Source

Each channel has a parameter base address in SPRAM, which is configured in ETPUCxCR registers, CPBA field, see Section 4.6.1, “eTPU Channel x Configuration Register (ETPUCxCR).” CHAN_BASE, which represents a parameter address (CPBA*8), can be used as A-source using T4ABS=1010 when T4ABS selects a source from the second register set. In this case, CHAN_BASE is loaded into AS[14:2] to form the byte address. For example, in indirect addressing mode, where the destination register is DIOB, CHAN_BASE is loaded into DIOB[14:2], which is the parameter address, and DIOB[14:0] represents the byte address. This is useful if the CHAN register is changed to transfer the base pointer to another channel.

9.3.3 Flags Sampling Control

This section explains how the flags Z (zero), C (carry), N (negative) and V (overflow) are updated in an ALU operation. When there are post-ALU shift operations, the ALU carry out is not directly sampled by the carry flag, but passed to the post-ALU shifter, see Section 9.3.7, “Shift Operations.” Since the size of source operands in ALU operations is variable, flags can be sampled as an operation of 8, 16 or 24 bits wide. The operation size selection is automatic, based on defined sizes of the sources and destination, using the equation:

```
operation_size = minor(size_of(destination),
        greater(size_of(A-Source), size_of(B-Source))
```

Operation size determination is illustrated by Table 9-12.

Table 9-12. Operation Size Determination

A Source	B Source	Destination	Operation Size
x	x	8 bits	8 bits
8 bits	8 bits	x	8 bits
16 or 24 bits	x	16 bits	16 bits
16 bits	8 or 16 bits	24 bits	16 bits
8 or 16 bits	16 bits	24 bits	16 bits
24 bits	x	24 bits	24 bits
x	24 bits	24 bits	24 bits

NOTE

Whenever BS = (constant) 0, its operation size is considered 8 bits, and all 24 bits in B-bus are set to 0. Therefore, all operations with BS = (constant) 0 have their size determined by AS and the destination only.

The CCS field (1 bit) controls whether flags will be updated or not (Table 9-13). When CCS bit exists in a microinstruction and CCS is set to 0, the operation size will be used to sample flags.

Table 9-13. Flag Sampling Using CCS field

CCS	Meaning
0	sample flags as defined by operation size
1	do not sample flags

In some microinstructions CCS field is replaced by CCSV (2 bits, Table 9-14). Flag sampling according to CCSV can be set as defined by the operation size, or fixed as 8 or 16 bit operations.

Table 9-14. Flag Sampling Using CCSV field

CCSV	Meaning
00	sample flags as an 8 bit operation
01	sample flags as a 16 bit operation
10	sample flags as defined by operation size
11	do not sample flags

When neither CCS nor CCSV are present in the microinstruction, flags are not sampled. CCS and CCSV do not affect the carry flag update on exchange bit operation, see Section 5.8.2.6, “Exchange Bit,” but do control the N and Z flag updates.

9.3.4 B-Source Inversion

The data selected as second source (T4BBS) can be inverted (bitwise boolean NOT) before the ALU operation. This is controlled by microinstruction field BINV (1 bit, Table 9-15). A zero value for BINV activates B-source inversion.

Table 9-15. B-Source Inversion (BINV)

BINV	Meaning
0	invert BS
1	keep BS bus unchanged

BINV also selects between ADC (addition with carry) or SBC (subtraction with borrow) enhanced ALU operation, with BS and carry flag inversion for SBC. Note that BINV does not invert the carry flag in fixed-carry operations (see Table 9-16).

When BINV = 0, T4BBS = 111 and CIN = 0, the value assigned to BS is 0x800000, instead of 0x0. See Section 9.3.6, “Generating “Max” Constant,” for more details.

9.3.5 Carry-in Control

The CIN field (1 bit, Table 9-16) controls the carry-in for addition/subtraction operations. The functionality of CIN field depends on the arithmetic operation selected by ALUOP. When ALUOP is not present in the microinstruction, the operation selected is ADD. For carry-in control in MDU operations, see Table 5-35.

Table 9-16. ALU Carry-In Control

operation	CIN=0	CIN=1
ADD (addition)	carry-in used is 1	carry-in used is 0
ADC (addition with carry)[1]	carry-in used is C flag	
SBC (subtraction with borrow)[2]	carry-in used is inverted C flag	

1. Selected by ALUOP=11000 and BINV=1
2. Selected by ALUOP=11000 and BINV=0

9.3.6 Generating “Max” Constant

When T4BBS = 111, CIN = 0 and BINV = 0, BS is assigned 0x800000 (called “max constant”) instead of 0x000000. “Max constant” is the value which, added to a time base value minus 1, gives the farthest wrapped time base value that satisfies a channel greater-equal comparison. See Section 5.2, “Enhanced Channels ((where should this go?? Own chp?? Seems best as subsection of another chp. Leave under Host Interface for now. -VG 5/2004))),” for more info.

9.3.7 Shift Operations

There are three types of shift operations:

- ALU
- post-ALU
- Shift register

Post-ALU and shift register are covered in the following sections. ALU shift operations are covered in Section 9.3.13, “ALU/MDU Operation Selection.”

9.3.8 Shift Register Operations

SR can be used as a general purpose register. SR can shift-right its contents by one bit and can be combined with a post-ALU shift operation. If field SRC (1 bit) in microcode is 0, SR will shift its contents 1 bit to the right according to the algorithmic description below. The SR shift operation also depends on the SHF or ALUOP fields. ALUOP and SHF never exist simultaneously in the same microinstruction format.

Table 9-17. Shift Register Control (SRC)

SRC	Meaning
0	SR shift right by 1 bit
1	do not shift

SR Operation:

```
SR[22:0] = SR[23:1];
if SHF == “01” or ALUOP == “10110” then
      SR[23] = ALU_OUT[0];
else
      SR[23] = 0;
```

```
endif;
```

9.3.9 Post-ALU Shift Operations

Post-ALU shifting can be selected by the SHF field (2 bits) or by some specific ALUOP field values. SHF and ALUOP fields are never both available in the same microinstruction format, i.e. they are mutually exclusive. Certain ALUOP combinations can specify non-post-ALU shifts.

Table 9-18. Post-ALU Shift Operation

Post ALU Operation	SHF[1]	ALUOP
shift left (1 bit)	00	10101
shift right (1 bit)	01	10110
rotate right (1 bit)	10	10111
no shift/rotate	11	any other[2]

1. ALU performs AS+BS before shift/rotate for all SHF values.
2. some ALUOP combinations perform shift/rotate, but not using the Post-ALU Shifter, see Table 9-23.

Post-ALU shift operations are performed regardless of the source and destination sizes. The carry flag is only updated when CCS or CCSV[1:0] fields allow it, see Section 9.3.3, “Flags Sampling Control.” An algorithmic description of post-ALU shift operations are presented below:

Shift left:

```
AD[23:1]=ALU_OUT[22:0];

AD[0] = 0;

/*
 * if flags can be updated (depends on CCSV/CCS) then C Flag =
 * either one of {ALU_OUT[23], ALU_OUT[15], ALU_OUT[7]};
 *See, Note:
 */
```

NOTE

ALU_Cout is the carry out for 24-bit operations.

Shift right, see Note: :

NOTE

For explanation about the SRC field see Table 9-17.

```
AD[22:0] = ALU_OUT[23:1];

if flags can be updated then C Flag = ALU_OUT[0]

if SRC field == “0” then
```

```
     AD[23] = ALU_Cout;
else
     AD[23] = 0;
endif;
```

Rotate right:

```
AD[22:0] = ALU_OUT[23:1];
AD[23] = ALU[0];
if flags can be updated then C Flag = adder_carry from bit 7, 15 or 23;
```

9.3.10 Conditional ALU/MDU Operation Execution

The 3-bit field AS/CE allows conditional execution of arithmetic operation, as shown in Table 9-19. The same field can also be used for overriding the size of A-source, see Section 9.3.11, “A-Source Size Override.”

Table 9-19. ALU/MDU Conditional Execution

AS/CE	Meaning
000	used for A-source size override, see Section 9.3.11, “A-Source Size Override.”
001	
010	execute if C = 1
011	execute if C = 0
100	execute if Z = 1
101	execute if Z = 0
110	execute if N = 1
111	execute unconditionally/no size override

Other non-ALU/MDU operations in the same microinstruction are not affected by the AS/CE field.

If a conditional operation is selected, there is **no** A-source size override; similarly, when size override for A-source is selected, the ALU/MDU operation executes unconditionally.

When a conditional ALU/MDU operation is not executed:

- the destination register is not updated.
- the ALU and MDU flags are not updated.
- MDU does not start any operation, i.e., MACH and MACL are not updated.
- SR does not shift.

- T4ABS-selected read-match does not occur.

9.3.11 A-Source Size Override

Some values if the AS/CE field are used for A-source size override, as shown in Table 9-20.

Table 9-20. A-Source Size Override

AS/CE	Meaning
000	A-source size override to 8 bits
001	A-source size override to 16 bits
010	used for conditional execution, see Section 9.3.10, “Conditional ALU/MDU Operation Execution.”
011	
100	
101	
110	
111	execute unconditionally/no size override

Register size override zero-pads an overridden source to 24 bits (if no sign extension is performed, see Section 9.3.12, “A-source Sign Extension,” and affects operation size calculation. When register source is wider than size override, most significant bits of selected register are not used as A source (zeros are used instead). When size override is used with MDU operations, it affects only the operand values, but not the operation size: MDU operation size is fully determined by the operation definition (fields ALUOP, ALUOPI).

Table 9-21. AS/CE field A source size override functionary

Size Override	Size of Selected Register	AS Value[1]
8 bits	8 bits	reg[7:0]
8 bits	16 bits	reg[7:0]
8 bits	24 bits	reg[7:0]
16 bits	8 bits	reg[15:0][2]
16 bits	16 bits	reg[15:0]
16 bits	24 bits	reg[15:0]

1. All values are zero-padded to 24 bits
2. Only reg[7:0] physically exists, reg[15:8] = 0x00

9.3.12 A-source Sign Extension

The SEXT microinstruction field forces sign extension of A source, see Table 9-22, according to the size of A operand, regardless of whether size override is specified by AS/CE field or not. The sign is taken from the size-overridden value, not the original one.

Table 9-22. A Source Sign Extension

SEXT	Meaning
0	extends sign of A source from its size to 24 bits
1	does not extend sign of A source

A-source sign is not extended in microinstructions without SEXT field, even if AS/CE field is present.

9.3.13 ALU/MDU Operation Selection

When field ALUOP is available in microinstruction, enhanced ALU operations shown in Table 9-23 can be performed, otherwise addition is performed. The ALU operations are defined in Section 5.8.2, “ALU and Post-ALU Shifter.” The MDU operations are defined in Section 5.8.3, “MAC and Divide Unit (MDU).”

Table 9-23. ALU Operation Selection (ALUOP)

ALUOP	Operation	Comment
00000	AS mults BS[7:0]	signed multiplication
00001	AS multu BS[7:0]	unsigned multiplication
00010	AS fmults BS[7:0]	signed fractional multiplication
00011	AS fmultu BS[7:0]	unsigned fractional multiplication
00100	AS mults BS[15:0]	signed multiplication
00101	AS multu BS[15:0]	unsigned multiplication
00110	AS fmults BS[15:0]	signed fractional multiplication
00111	AS fmultu BS[15:0]	unsigned fractional multiplication
01000	AS mults BS[23:0]	signed multiplication
01001	AS multu BS[23:0]	unsigned multiplication
01010	AS macs BS[23:0]	signed multiply-accumulate
01011	AS macu BS[23:0]	unsigned multiply-accumulate
01100	AS div BS[7:0]	unsigned division by 8-bit value
01101	AS div BS[15:0]	unsigned division by 16-bit value
01110	AS div BS [23:0]	unsigned division by 24-bit value
01111	n.a.	RESERVED

Table 9-23. ALU Operation Selection (ALUOP)

ALUOP	Operation	Comment
10000	AS[23:0] | BS[23:0]	24 bit bitwise OR
10001	AS[23:0] ^ BS[23:0]	24 bit bitwise XOR
10010	AS[23:0] & BS[23:0]	24 bit bitwise AND
10011	abs(AS)	absolute value of AS
10100	AS + BS	arithmetic addition
10101	(AS + BS) shl 1	arithmetic addition with 1-bit post-ALU shift left. Section 9.3.9, “Post-ALU Shift Operations.”
10110	(AS + BS) shr 1	arithmetic addition with 1-bit post-ALU shift right, Section 9.3.9, “Post-ALU Shift Operations.”
10111	(AS + BS) ror 1	arithmetic addition with 1-bit post-ALU rotate right, Section 9.3.9, “Post-ALU Shift Operations.”
11000	AS adc/sbc BS[1]	addition/subtraction with C flag, Section 9.3.5, “Carry-in Control.”
11001	AS shl (2^(BS[1:0]+1))	AS is shifted left: 2 bits for BS=0; 4 for BS=1; 8 for BS=2; 16 for BS=3
11010	AS shr (2^(BS[1:0]+1))	AS is shifted right: 2 bits for BS=0; 4 for BS=1; 8 for BS=2; 16 for BS=3
11011	AS ror (2^(BS[1:0]+1))	AS is rotated right: 2 bits for BS=0; 4 for BS=1; 8 for BS=2; 16 for BS=3
11100	AS EXCH BS[4:0]	exchange C flag and AS bit determined by BS[4:0], Section 5.8.2.6, “Exchange Bit.”
11101	AS SETB BS[4:0]	set bit in AS determined by BS[4:0] [2]
11110	AS CLRB BS[4:0]	clear bit in AS determined by BS[4:0][2]
11111	n.a.	RESERVED

1. Addition/subtraction is selected by field BINV, see Section 9.3.4, “B-Source Inversion.”
2. in SETB and CLRB operations, the register that drives A source is not changed, unless selected as destination of the operation.

9.3.14 Operations With Immediate Data

Immediate data can be used with some specific microinstruction formats. eTPU microcode allows 8-, 16- or 24-bit immediate data. Immediate data is loaded as B-source, so T4BBS field is not available. When using 24-bit immediate data, an add is performed with A-source = 0 and the ALU flags are not updated. ALU operations are available with 8-bit immediate data, although the field that selects ALU operation in this case is ALUOPI.

9.3.14.1 24-bit Immediate Destination

When using 24-bit immediate data, the destination register is selected by T2D field (2 bits), according to Table 9-24.

Table 9-24. 24-bit Immediate Destination (T2D)

T2D	Target Register
00	P[23:0]
01	A[23:0]
10	SR[23:0]
11	DIOB[23:0]

9.3.14.2 Enhanced ALU Operations With Immediate Data

Enhanced operations with immediate data, selected by ALUOPI (5 bits) are allowed only with an 8 bit immediate operand (see Table 9-25).

Table 9-25. ALU Operation Selection With Immediate Data (ALUOPI)

ALUOPI	Operation	Comment
00000	AS mults #imm8	signed multiplication
00001	AS multu #imm8	unsigned multiplication
00010	AS fmults #imm8	signed fractional multiplication
00011	AS fmultu #imm8	unsigned fractional multiplication
00100	AS div #imm8	unsigned division
00101	n.a.	reserved
00110	n.a.	reserved
00111	n.a.	reserved
01000	AD[7:0] = AS[7:0] | #imm8, AD[23:8] = AS[23:8]	bitwise OR
01001	AD[7:0] = AS[7:0] ^ #imm8, AD[23:8] = AS[23:8]	bitwise XOR
01010	AD[7:0] = AS[7:0] & #imm8, AD[23:8] = AS[23:8]	bitwise AND
01011	AD[7:0] = AS[7:0] & #imm8, AD[23:8] = 0x0	bitwise AND with clear
01100	AD[15:8] = AS[15:8] | #imm8, AD[23:16] = AS[23:16], AD[7:0] = AS[7:0]	bitwise OR
01101	AD[15:8] = AS[15:8] ^ #imm8, AD[23:16] = AS[23:16], AD[7:0] = AS[7:0]	bitwise XOR
01110	AD[15:8] = AS[15:8] & #imm8, AD[23:16] = AS[23:16], AD[7:0] = AS[7:0]	bitwise AND

Table 9-25. ALU Operation Selection With Immediate Data (ALUOPI) (continued)

ALUOPI	Operation	Comment
01111	AD[15:8] = AS[15:8] & #imm8, AD[23:16] = 0x0, AD[7:0] = 0x0	bitwise AND with clear
10000	AD[23:16] = AS[23:16] | #imm8, AD[15:0] = AS[15:0]	bitwise OR
10001	AD[23:16] = AS[23:16] ^ #imm8, AD[15:0] = AS[15:0]	bitwise XOR
10010	AD[23:16] = AS[23:16] & #imm8, AD[15:0] = AS[15:0]	bitwise AND
10011	AD[23:16] = AS[23:16] & #imm8, AD[15:0] = 0x0	bitwise AND with clear
10100	AS + #imm8	arithmetic addition
10101	(AS + #imm8) shl 1	arithmetic addition with 1-bit shift left.
10110	(AS + #imm8) shr 1	arithmetic addition with 1-bit shift right
10111	(AS + #imm8) ror 1	arithmetic addition with 1-bit rotate right
11000	n.a.	reserved
11001	AS shl (2^(#imm8[1:0]+1))	AS is shifted left: 2 bits for #imm8=0; 4 for #imm8=1; 8 for #imm8=2; 16 for #imm8=3
11010	AS shr (2^(#imm8[1:0]+1))	AS is shifted right: 2 bits for #imm8=0; 4 for #imm8=1; 8 for #imm8=2; 16 for #imm8=3
11011	AS ror (2^(#imm8[1:0]+1))	AS is rotated right: 2 bits for #imm8=0; 4 for #imm8=1; 8 for #imm8=2; 16 for #imm8=3
11100	AS exch #imm8[4:0]	exchange C flag and AS bit determined by #imm8[4:0], see Section 5.8.2.6, "Exchange Bit."
11101	n.a.	reserved
11110	n.a.	reserved
11111	n.a.	reserved

9.4 Channel Control and Configuration Microoperations

Channel control and configuration fields set configuration values, except for the LSR and CIRC fields, in the channel logic of a specified channel. The channel is specified by the CHAN register.

9.4.1 Channel Flags Operations

Each channel has two associated hardware flags, called Channel Flag 0 and Channel Flag 1. The microcode field FLC (3 bits) allows the flags to be set or cleared, as shown in

Table 9-26. These flags cannot be directly tested by microcode, but may influence the channel function's entry point. The channel flags provide conditional information required by the entry point table and allow fast state decoding. For more details, see Section 5.1.1, "Entry Points." The channel flags may be directly set by the FLC register or copied from bit fields in the P register.

Table 9-26. P Flags Operation (FLC)

FLC	Meaning
000	clear flag0
001	set flag0
010	clear flag1
011	set flag1
100	copy flag1:flag0 from P[25:24]
101	copy flag1:flag0 from P[27:26]
110	copy flag1:flag0 from P[29:28]
111	no operation (nil)

9.4.2 Comparator and Time Base Selection

TBS1 and TBS2 fields (4 bits wide each) are used to configure the type of the comparator and the time bases used for match or capture, see Table 9-27 and Table 9-28).

Table 9-27. Time Base Selection 1 (TBS1)

	TBS1 bit	**2**	**1**	**0**
TBS1[3] = 0	**Bitfield**	**Comparator selection**	**Capture selection**	**Match TB selection**
	0	greater or equal	TCR1	TCR1
	1	equal-only	TCR2	TCR2
TBS1[3] = 1	**action**	**2**	**1**	**0**
	Do nothing	1	1	1
	Reserved	All other values		

Table 9-28. Time Base Selection 2 (TBS2)

TBS2[3] = 0	**TBS2 bit**	**2**	**1**	**0**
	bitfield	**Comparator selection**	**Capture selection**	**Match TB selection**
	0	Greater or equal	TCR1	TCR1
	1	Equal-only	TCR2	TCR2
TBS2[3] = 1	**action**	**2**	**1**	**0**
	Do nothing	1	1	1
	Reserved	All other values		

9.4.3 Transition Detection and Pin Action Control

IPAC1/2 and OPAC1/2 fields are used to configure transition detection sensitivity (for the channel input signal) or output pin action control (for the channel output signal), as defined in Table 9-29. IPAC1 and IPAC2 have the same format, where IPAC1 is related to Match1 and first transition detection, and IPAC2 to Match2 and second transition detection. The same applies in analogue way to OPAC1 and OPAC2.

For the output signal, configuring OPAC registers doesn't change the current signal state, but defines the action to be done when a match or input action occurs. See Section 5.2.2, "Match Recognition," and Section 5.2.4.24, "Channel Modes on Output Signal Generation," for more information. IPAC1/2=1xx also enables assertion of MRL_A/B during time slot transition, see Section 5.2.2, "Match Recognition."

Table 9-29. Input and Output Pin Action Control (IPAC1/2) and (OPAC1/2)

value	IPAC meaning	OPAC meaning
000	Do not detect transitions	Do not change output signal
001	Detect rising edge only	Match[1] sets output signal high
010	Detect falling edge only	Match[1] sets output signal low
011	Detect both edges	Match[1] toggles output signal
100	Detect input signal = 0 on Match[1]	Input action sets output signal low
101	Detect input signal = 1 on Match[1]	Input action sets output signal high
110	Reserved	Input action toggles output signal
111	Do not change IPAC	Do not change OPAC

1. Match 1 is used for IPAC1/OPAC1, and Match 2 for IPAC2/OPAC2.

9.4.4 Immediate Pin State Control

It is possible to change output signal state immediately by using PSC (2 bits) and PSCS (1 bit) fields.

Table 9-30. Immediate Pin State Control (PSC) and (PSCS)

PSC	PSCS	Meaning
00	0	Set signal as specified by OPAC1, see Section 9.4.3, "Transition Detection and Pin Action Control."
00	1	Set signal as specified by OPAC2, see Section 9.4.3, "Transition Detection and Pin Action Control."
01	x	Set signal high
10	x	Set signal low
11	x	Don't change signal state

9.4.5 Write Channel Match Registers

Match registers can have their values changed using ERW1 and ERW2 fields (1 bit each). ERW1/2 also set their respective MRLE registers, see Section 5.2.2, "Match Recognition."

Table 9-31. Write Match1/2 (ERW1/2)

Field	Value	Action
ERW1	0	Write ERT_A value in Match1. Enable matches for Match1 register (MRLE1=1)
	1	Don't change Match1 and MRLE1
ERW2	0	Write ERT_B value in Match2. Enable matches for Match2 register (set MRLE2=1)
	1	Don't change Match2 and MRLE2

If ERT_A or ERT_B is a destination of an ALU operation and, at the same time, the respective ERW1/2 field is active, the new ERT_A value is the one written into the Match1/2 register.

9.4.6 Clear Transition/Match Event Registers

Flags MRL_A, MRL_B, TDL_A and TDL_B, see Section 5.2.1.1, "Event Registers (ER)," indicate the state of matches and transitions detected in the selected channel. It is possible to clear those flags using the microcode fields MRL_A, MRL_B and TDL (1 bit each), see Note: .

NOTE

One bit, TDL, is used to clear both TDL_A and TDL_B flags.

The flags cleared by these microcode fields are the actual channel flags as well as the flags sampled into the branch logic.

Table 9-32. Clear Transition/Match Event Registers (MRL_A/B), (TDL)

Field	Meaning
MRL_A	0 = clear MRL_A event register, 1 = don’t change
MRL_B	0 = clear MRL_B event register, 1 = don’t change
TDL	0 = clear TDL_A and TDL_B flags, 1 = don’t change

9.4.7 Disable Matches

The microcode field MRLE (1 bit) allows disabling matches on the channel selected by CHAN register, for both Match1 and Match2 registers. Matches can be enabled for each match register using ERW1 and ERW2 fields, see Section 9.4.5, “Write Channel Match Registers.”

Table 9-33. Disable Matches (MRLE)

MRLE	Meaning
0	Disable matches for Match1 and Match2 registers
1	don’t change match enabling

9.4.8 Disable Match and Transition Service Requests

Microcode field MTD (2 bits) disables match and transition service requests for the selected channel. MTD does not disable link service request and host service request. MTD sets or resets register SRI, for more details see Section 5.2.1.4.3, “Match/Transition Service Request Inhibit Latch (SRI).”

Table 9-34. Disable Match and Transition Service Request (MTD)

MTD	Meaning
00	SRI = 0: enable service requests for match and transition
01	SRI = 1: disable service requests for match and transition
10	reserved
11	don’t change

9.4.9 Predefined Channel Modes

The PDCM field (4 bits) in eTPU microcode defines the channel mode, see Section 5.2.4, “Channel Modes.”

PDCM coding is shown in Table 9-35. Note that PDCM bit 0 selects between single transition (PDCM[0]=0) and double transition (PDCM[0]=1) modes.

Table 9-35. Predefined Channel Modes

PDCM	Channel mode
0000	em_b_st
0001	em_b_dt
0010	em_nb_st
0011	em_nb_dt
0100	m2_st
0101	m2_dt
0110	bm_st
0111	bm_dt
1000	m2_o_st
1001	m2_o_dt
1010	reserved
1011	reserved
1100	sm_st
1101	sm_dt
1110	sm_st_e
1111	keep current channel mode

9.4.10 Channel Interrupt and Data Transfer Requests

eTPU microcode can issue interrupt requests, data transfer requests and a global exception through CIRC field. CIRC affects the serviced channel. For more information see Section 5.0.2, “Interrupts and Data Transfer Requests.”

Table 9-36. Channel and Data Transfer Requests (CIRC)

CIRC	Meaning
00	Channel Interrupt Request
01	Data Transfer Request
10	Global Exception
11	don’t request interrupt

9.4.11 Clear Link Service Request

The LSR microcode field (1 bit) is used to clear the link service request flag of the serviced channel (may not be the one selected by CHAN). The LSR branch condition is always cleared, but not the link service request, if another channel link was received by the serviced channel during the executing thread. See Section 5.2.5, “Channel Link,” for more information.

Table 9-37. Link Service Request Negation Control (LSR)

LSR	Meaning
0	clear link service request (flag LSR)
1	don’t change

9.5 Flow Control Microoperations

The eTPU has jump and call microoperations to control the microcode flow. In addition the eTPU has dispatch jump and dispatch call operations which can be used to implement a jump table. In the call (or dispatch call) microoperation, the return address is saved in RAR register. If nested sub-routine calls are necessary, the return address values have to be saved in a stack, usually implemented with the DIOB register.

Flow control microoperations are also provided to finish the current thread execution and to halt the microengine.

9.5.1 Ending Current Thread (END)

The microcode END field (1 bit) finishes current thread and allows other channels to be serviced. If END field is 0, the current instruction is completed and the thread is finished. END = 1 has no effect, and the next microinstruction in the thread is executed. Any MDU operation, see Section 5.8.3, “MAC and Divide Unit (MDU),” that could be still pending when the thread is finished is left incomplete. END also releases any semaphore locked by the engine.

9.5.2 Branch Operations

Branch operations can be jump or call. The target address of jump or call microoperations are always immediate and absolute. The branch microoperation is affected by FLS field, refer to Section 9.5.5, “Flush Pipeline.”

9.5.2.1 Selecting Jump or Call Microoperations

The only difference between jump and call microoperations is that when a call is executed the value of PC is saved in RAR register. The microcode field J/C (1 bit) selects whether jump or a call is executed, according to Table 9-38.

Table 9-38. Jump / Call Selection (J/C)

J/C	Meaning
0	jump
1	call

9.5.2.2 Branch Target Address

The BAF microcode field (14 bits) indicates the absolute address of a jump/call target.

9.5.2.3 Conditional/Unconditional Branch

Jump and call can be conditional or unconditional, depending on the BCF (1 bit) fields and BCC (5 bits), as shown in Table 9-39 and Table 9-40. BCF determines branching based upon whether the condition specified by BCC is true or false. When a branch condition uses the channel flags, the channel context is related to the channel number written in CHAN register.

Table 9-39. Branch Condition Inversion (BCF)

BCF	Meaning
0	branch if condition determined by BCC is false
1	branch if condition determined by BCC is true

Table 9-40. Branch Condition Selection (BCC)

BCC	Meaning
00000	V ALU flag
00001	N ALU flag
00010	C ALU flag
00011	Z ALU flag
00100	MV MDU flag
00101	MN MDU flag
00110	MC MDU flag
00111	MZ MDU flag
01000	TDL_A channel flag
01001	TDL_B channel flag
01010	MRL_A channel flag
01011	MRL_B channel flag
01100	LSR channel flag
01101	MB flag MDU flag

Table 9-40. Branch Condition Selection (BCC) (continued)

BCC	Meaning
01110	FM[1] channel flag
01111	FM[0] channel flag
10000	PSS channel flag
10001	reserved
10010	“Less Than” ALU flag combination (signed) [1]
10011	“Lower or Equal” ALU flag combination (unsigned)[2]
10100	P[24]
10101	P[25]
10110	P[26]
10111	P[27]
11000	P[28]
11001	P[29]
11010	P[30]
11011	P[31]
11100	PSTO channel flag
11101	PSTI channel flag
11110	SMLCK semaphore flag
11111	false

1. “less than” is the xor between ALU flags V and N.
2. “lower equal” is Z or not C.

9.5.3 Dispatch Microoperation

The dispatch microoperation is an unconditional branch where the target address is always PC+P[31:24] (unsigned). Dispatch is affected by FLS field, refer to Section 9.5.5, “Flush Pipeline.” Dispatch is defined by R/D field (2 bits, Table 9-41). The R/D field can also be used to define return from sub-routine, see Section 9.5.4, “Return from Subroutine.”

Table 9-41. Return and Dispatch (R/D)

R/D	Meaning
00	return from subroutine, see Section 9.5.4, “Return from Subroutine.”
01	dispatch jump
10	dispatch call
11	don’t change microinstruction flow

9.5.4 Return from Subroutine

When a subroutine call or a dispatch call microoperation is executed, the return address is saved in RAR register. To return from a call subroutine, a microoperation loads the contents of the RAR register back to the PC. Fields R/D (2 bits, Table 9-41) or RTN (1 bit, Table 9-42) can be used to return from subroutine.

The return from subroutine microoperation is affected by FLS, see Section 9.5.5, “Flush Pipeline,” when field R/D is used. Execution of return from subroutine through RTN always flushes the pipeline.

Table 9-42. Return from Sub-routine (RTN)

RTN	Meaning
0	return with pipeline flush
1	do not return

9.5.5 Flush Pipeline

When a branch, dispatch or subroutine return microoperation is executed, the next microinstruction can be executed unconditionally before the flow change takes effect, since microengine has a two-stage pipeline. Executing the next microinstruction after a branch **maximizes** execution performance. This feature is controlled by field FLS (1 bit, Table 9-43). When FLS=0 the pipeline is flushed and the next microinstruction placed after a branch is decoded as NOP if the branch is taken. If FLS=1, the microinstruction placed after the branch is executed, either if the branch is taken or not, as shown in Figure 9-1.

Table 9-43. Flush Pipeline (FLS)

FLS	Meaning
0	flush pipeline when jump / call / dispatch jump / dispatch call / return is executed
1	do not flush pipeline when jump / call / dispatch jump / dispatch call / return is executed

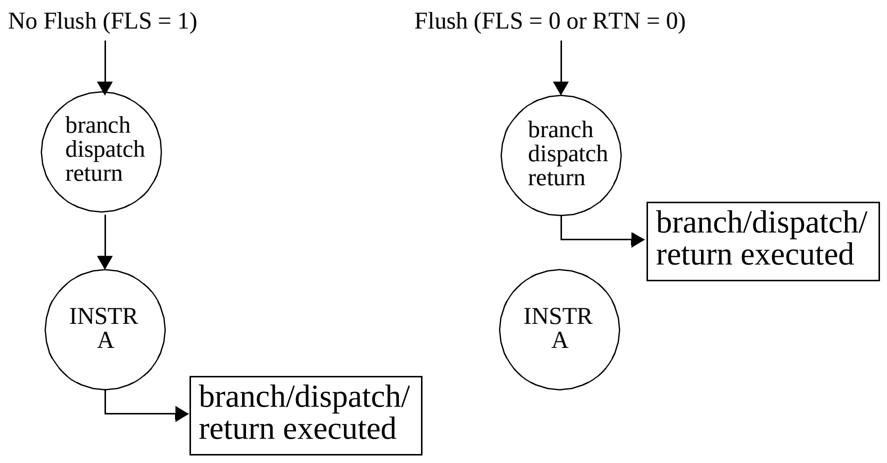

No Flush (FLS = 1)
branch dispatch return
INSTR A
branch/dispatch/ return executed
Flush (FLS = 0 or RTN = 0)
branch dispatch return
branch/dispatch/ return executed
INSTR A

Figure 9-1. Flush Pipeline

9.5.6 HALT Microinstruction

HALT is a microinstruction provided to implement software breakpoints, see Section 5.10.2.5, “Software Breakpoints.” Note that HALT is coded as a microinstruction format, not a field, see Section 9.8, “Microinstruction Formats.” The execution of this instruction puts the microengine in halt state. For more information about the implications of microengine halt state, see Section 5.10.2.2, “Microengine Halt State.” HALT is valid only if the debug mode is enabled at the debug interface. If debug is not enabled, HALT executes as a NOP and is treated as an illegal instruction, see Section 9.6, “Illegal Instructions.”

9.5.7 NOP Microinstruction

There is not a unique microinstruction with an assigned opcode to do No Operation. NOP microinstruction is achieved through any of the formats shown on Table 9-45 where the user can assign to each individual field the corresponding value for “No Operation”. However, to prevent future impacts of instruction changes on object code compatibility, the instruction value 0x4FFFFFFF should always be used for NOP.

9.6 Illegal Instructions

An instruction is considered illegal if any reserved field value is used, including zero bits at the fields marked rsv in the instruction formats (see Table 9-45). A global exception may be issued up to two microcycles after instruction fetch. The execution results of an illegal instruction on the microengine, channel logic or host interface are unpredictable.

If the microengine decodes an illegal instruction, the following actions are taken:

- A global exception is issued.
- Flag ILFA/B on register ETPUMCR is set to indicate this occurrence to the host.

9.7 Microinstruction Parallelism Issues

This section clarifies parallelism issues that arise when two non-commutative microoperations appear in the same microinstruction.

9.7.1 ALU Operations and Read Match Registers

ALU operations have only one destination register, but there is one case where source selection determines destination: copy match registers to ERT_A and ERT_B registers. In this case if the ALU destination is ERT_A or ERT_B a conflict arises. The ALU destination value overwrites the value read from the match registers.

9.7.2 ALU and SPRAM Operations

P and DIOB registers can be selected as a destination by both ALU and SPRAM (read) microoperations in the same microinstruction. Since P and DIOB update from SPRAM data happens after P and DIOB update for ALU/MDU microoperations, the data read from SPRAM will overwrite any results from ALU/MDU microoperations in P or DIOB when either of them is specified as destination for both an ALU and SPRAM microoperation. However, the ALU operation is executed and its flags are updated accordingly. All the above also applies to zero SPRAM operations.

If DIOB is the ALU destination and P is loaded from SPRAM or vice-versa, no conflict occurs, and the result is the same as if operations occurred separately.

When using P or DIOB as destination for ALU operations and also as source for a SPRAM write operation, the data written in SPRAM is the one calculated by the current ALU operation, which means it is possible to calculate a value and write it in an SPRAM address using only one microinstruction.

9.7.3 ERT_A/B as ALU destination and ERW1/2

The value in ERT_A and ERT_B registers can be written to the match registers of the selected channel by using fields ERW1 and ERW2, Section 9.4.5, “Write Channel Match Registers.” If during the same microinstruction ERT_A or ERT_B is destination of an ALU/MDU microoperation, the value written in match registers is the ALU/MDU result.

If an ALU operation occurs in parallel with ERW1/2 but ERT_A/B are not the destination of an ALU/MDU operation, then Match1/2 receives the original ERT_A/B value.

9.7.4 ERW1/2 and MRLE

ERW1/2 automatically set the MRLE1/2 channel latches, respectively, see Section 9.4.5, “Write Channel Match Registers.” The microinstruction fields ERW1/2 independently set MRLE1/2 channel flags, regardless of any MRLE microinstruction field value.

9.7.5 CHAN Assignment, Read Match and ERW1/2

When CHAN is a destination of an ALU operation it causes a read of the Capture1/2 register values into ERT_A/B. The capture registers loaded into ERT_A/B are selected by the new CHAN value. The value of the Capture1/2 registers overwrites any read-match commanded simultaneously.

If CHAN assignment happens with an ERW1/2 operation in the same instruction, the updated match register(s) belong to the new selected channel.

9.7.6 Read Match and ERW1/2

If a read match operation is executed with ERW1/2 in the same microinstruction, the Match1/2 registers receive the old values of ERT_A/B, and the ERT_A/B registers receive the old Match1/2 values simultaneously, i.e.: ERT_A/B and Match1/2 swap their values.

If ERT_A/B is the destination of an ALU operation at the same instruction, Match1/2 gets the ALU result, see Section 9.7.3, “ERT_A/B as ALU destination and ERW1/2,” but ERT_A/B still receives the old Match1/2 values.

NOTE

Read match, ERW1/2 and CHAN assignment can be active at the same instruction. Combining the rules from Section 9.7.5, “CHAN Assignment, Read Match and ERW1/2,” and Section 9.7.6, “Read Match and ERW1/2,” the result is: ERT_A/B receives the Capture1/2 values of the new CHAN value, and Match1/2 receives the old ERT_A/B value(s).

9.7.7 Stack Accesses and ALU Operations

Post-increment is ignored in a stack operation (field STC) if DIOB is loaded from SPRAM: DIOB keeps the value read from SPRAM. Pre-decrement is ignored in a stack operation (field STC) if DIOB is destination of an ALU operation. Post-increment/pre-decrement remains valid in all other situations. These rules are summarized in Table 9-44.

Table 9-44. DIOB load from SPRAM and ALU

DIOB load from SPRAM and ALU	DIOB selected as ALU destination?	DIOB Load Value
no	no	DIOB, --DIOB (pre-decrement), or DIOB++ (post-increment)
yes	no	SPRAM read data (post-inc and pre-dec ignored)
yes	yes	SPRAM read data (post-inc, pre-dec and ALU result ignored)
no	yes	ALU result (post-inc an pre-dec ignored)

9.7.8 SRC and ALU Operations

If operation SRC is active (field SRC = 0) and register SR is selected as destination of an ALU operation, the value of the ALU operation prevails over the original SR shifted value and is loaded into the SR register.

9.7.9 Semaphore Lock/Free and SMLCK Branch Condition

When the SMLCK branch condition is tested at the same microinstruction of a semaphore lock or free, the condition is evaluated after the semaphore action (either free or lock) is taken.

9.8 Microinstruction Formats

Table 9-45. Microinstruction Formats

format		microinstruction																															
new	**old**	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
A1	5C	0 0 0			IMM[15:13]			IMM[7:2]						IMM[23:16]								IMM[12]	RTN	IMM[11:9]			IMM[1:0]		T2D		IMM[8]	0 0	
A2	5B													T4ABS				T2ABD					CCS						ABSE	ABDE		0 1	
A3	5D				ALUOPI[4]	CCSV																ALUOPI[3:2]		AS/CE					ALUOPI[1:0]		0	1 0	
A4	5A				FLC[2]	SHF																SRC	CCS	rsv	FLC[1:0]				ABSE	ABDE	1		
B1	1A	1 0		0	END			CIN	BINV	T4BBS			RW										P/D		CCS	AID[7:0] (global param)							
B2	1C			1																						ZRO	AID[6:0] (channel param)						
B3	1D	0 0 0																									STC		ABSE	ABDE	rsv	1 1	
B4	1E	0 0 1			0	CCSV																1		AS/CE					ALUOP				
B5	1F												FL									0	SEXT				SMPR						
B6	1G				1								rsv									SRC					ABSE	ABDE					
B7	2	0 1 1			END	SHF							TDL										PSC		MRL_A	ERW1	MRL_B	ERW2	ABSE	ABDE	CCS	MRLE	PSCS

Table 9-45. Microinstruction Formats (continued)

format		microinstruction																															
new	old	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C1	3H	0 1 0			0	END	OPAC1			OPAC2				TBS1				TBS2				LSR	PSC		MRL_A	ERW1	MRL_B	ERW2	PDCM				PSCS
C2	3I				1		IPAC1			IPAC2																							
D1	3A	1 1 0			0		MRLE	rsv		PSC		FLS	RW	PSCS	FLC			CIRC		R/D		0	P/D	RSIZ		AID[7:0] (global param)							
D2	3C																					1				ZRO	AID[6:0] (channel param)						
D3	3DE	1 1 1			rsv																	1					STC		1 1		0	0	rsv
D4	3F												FL									0	rsv				SMPR						
D5	4A	1 1 0			1					MTD		rsv	RW	TDL				MRL_A	ERW1	MRL_B	ERW2	0	P/D	RSIZ		AID[7:0] (global param)							
D6	4C																					1				ZRO	AID[6:0] (channel param)						
D7	4DE	1 1 1			rsv																	1					STC		1 1		0	1	rsv
D8	4F												FL									0	rsv				SMPR						
E1	3G1	1 1 1			rsv	J/C	BCC					FLS	RW	BCF	BAF[13:0]														00		P/D	STC	
E2	3G2																												01		AID[2:0]		
E3	3G3												FL																10		rsv	SMPR	
E4	3G4												0																11		1	rsv	
F1	HALT				rsv								1	rsv															111			rsv	
format		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

all bits marked *rsv* are reserved, and must be coded as 1.

Execution Unit Operations	Channel Control Operations
RAM Input/Output Operations	Microengine/Sequence Operations

Chapter 10 Test and Development Support

10.1 Introduction

This chapter describes several development and test features available on the eTPU. Most debug features, described in Section 10.2, "Development Support Features," are accessible through a separate debug bus; refer to Chapter 11, "Nexus Dual eTPU Development Interface (NDEDI)," for more information.

Section 10.3, "Test Support Features," describe embedded test features: the multiple input signature calculator (MISC) is an SCM test feature accessible through registers ETPUMCR and ETPUMISCCMPR, see Section 4.2, "System Configuration Registers." MISC allows SCM test "on the fly", i.e., while eTPU is running, with no impact on eTPU functionality or performance.

10.2 Development Support Features

10.2.1 Internal Debug Interface and Nexus Class 3 Support

The eTPU provides an internal debug interface that exports real-time microengine states and values, including breakpoint/watchpoint information. It also provides inputs for breakpoint requests from other on-chip peripherals or off-chip devices. Refer to Chapter 11, "Nexus Dual eTPU Development Interface (NDEDI)," for more information.

10.2.2 Microengine Halt State

Halt is a microengine state where it stops executing during a thread, or does not start executing a scheduled thread from idle state. While idle state is entered from END execution without any other scheduled thread, microengine enters halt state by any of the following events:

- Execution of the HALT microinstruction (software breakpoint).
- External halt request through the debug interface (includes Nexus breakpoint request via EVTI input pin; see Chapter 11, "Nexus Dual eTPU Development Interface (NDEDI)," for more information)

- The other engine enters halt state and they are configured to halt simultaneously (bit HTWIN is asserted via the Nexus interface).
-
- Occurrence of any of the hardware breakpoint conditions. See Section 10.2.3, “Hardware Breakpoints,” for details.
- Execution of a single-step microinstruction: microengine returns to halt state after executing a single microinstruction while in halt state. See Section 10.2.6, “Single-step Execution,” and Section 10.2.7, “Forced Microinstruction Execution,” for details.

The eTPU MDU continues executing until it finishes any ongoing operation even if microengine is in halt state except when the halted instruction is an END.

There are two kinds of halt state, depending on what it was doing when halted:

- **halt_idle**, if the engine was not executing a thread when halted; the engine cannot leave halt_idle to fetch instructions, so one cannot single-step or follow a program flow; it can, however, execute forced instructions (see Section 10.2.7, “Forced Microinstruction Execution”).
- **halt_exec**, if the engine was executing a thread when halted. The engine can single-step and continue a program flow from halt_exec.

When the microengine exits halt state, the instruction pointed by the PC is fetched, while the instruction already fetched before halt is executed. Note that both the PC and the pre-fetched instructions can be modified during halt state, with a forced execution of a branch instruction, see Section 10.2.7, “Forced Microinstruction Execution.”

10.2.3 Hardware Breakpoints

The microengine can enter halt state through a command from the debug interface, configuring a hardware breakpoint. Hardware breakpoints halt the microengine on specific conditions. Occurrence of any of the following conditions can halt the microengine, that is they can be individually enabled.

- CHAN register assignment (only by microcode, not by time slot transition).
- SPRAM read and/or write to a given address and/or write data. The breakpoint is always qualified by the SPRAM address, but the following variations are allowed:
- Break on write only, read only, or read-and-write.
- Break on higher-byte write data value, lower 24-bit write value, full word (32-bit) write value, or regardless of data. Break on read data is not supported.
- PC (program counter) value.
- Beginning of a thread with a host service request pending.

- Beginning of a thread with a link service request pending.
- Beginning of a thread with a match service request pending.
- Beginning of a thread with a transition service request pending.

All these conditions can also be qualified by the value of the CHAN register.

On any of these conditions the halt of one microengine is independent of the other, unless the other engine is configured to halt simultaneously via Nexus Interface.

While in halt state, the microengine can also execute in single-step, see Section 10.2.6, "Single-step Execution," or any forced microinstruction not in the normal program flow, see Section 10.2.7, "Forced Microinstruction Execution."

10.2.4 Hardware Watchpoints

Debug Interface allows for watchpoints on the same conditions available for hardware breakpoints, see Section 10.2.3, "Hardware Breakpoints."

10.2.5 Software Breakpoints

A software breakpoint occurs when microengine executes a HALT microinstruction. Any number of software breakpoints can be set in code, usually replacing an active microinstruction.

Like any other microinstruction, HALT increments the PC and pre-fetches the next instruction. So, before the halt state is suspended, if the original program flow must be followed, the original instruction at the HALT address must be forced through the debug interface and executed, see Section 10.2.7, "Forced Microinstruction Execution," regardless if the software breakpoint is removed (replacing HALT by the original microinstruction) or not.

Special care must be taken if HALT is followed by another HALT, and the second HALT is removed from the SCM when the microengine was halted by the first one. In this case, replacing the second HALT by the original microinstruction is not enough to remove the second breakpoint, because the second HALT is already pre-fetched and will be executed when halt is suspended. To resolve the aforementioned issue, the debugger must also do a forced execution of unconditional branch with flush to the original microinstruction address. That will clear the pipeline, replacing the prefetched instruction by a NOP, and load PC with the address of the removed breakpoint. So, when halt state is suspended, the original microinstruction will be fetched while NOP is executed, and program flow continues normally from then on.

There is only one way of inserting software breakpoints into SCM RAM: writing bit VIS=1 in register ETPUMCR, and then accessing SCM as an ordinary RAM. This can be done only if both engines are halted or stopped.

10.2.6 Single-step Execution

When the microengine is already in halt state, it can run the next microinstruction in the normal program flow and get back to halt state. PC is incremented, or assigned the BAF value in a branch with a satisfied condition. Note that the executed instruction was already pre-fetched in the instruction pipeline, and a new microinstruction is fetched during its execution. The pre-fetched instruction may be cleared during halt state by the forced execution of a branch (or an unconditional branch) with flush, see Section 10.2.7, “Forced Microinstruction Execution,” making single-step execute a NOP instead of the next instruction in the program flow.

Single-step execution is controlled by the debug interface, and is a feature available from Nexus. The single-step execution of a NOP instruction is useful to control input signal sampling and filtering.

10.2.7 Forced Microinstruction Execution

When microengine is already in halt state, it can run forced microinstructions through the Nexus debug interface. The microinstruction, specified by the user, is not fetched from the SCM and comes directly from the debug interface. MDU start commands issued by forced instructions are ignored. The microinstruction field END is also ignored.

During forced execution of any instruction except branches, returns and dispatches, the PC does not change, and previous pre-fetched instruction in the pipeline is bypassed, but not discarded. When halt state is suspended, the pre-fetched instruction is executed and the instruction pointed by the PC is pre-fetched in parallel (two-stage pipeline).

Forced execution of a branch loads the PC with the BAF field if branch condition is satisfied. If branch condition is not satisfied, the PC value stays unaltered. The flush control (field FLS) also works, so that a successful forced branch with flush replaces the pre-fetched instruction by a NOP. Therefore, an unconditional branch to the desired address with flush is all one has to do to clear the instruction pipeline during halt.

Forced operations that depend on the serviced channel are unpredictable when executed in halt_idle.

10.2.8 Microengine Register Access

The eTPU does not provide direct access to the microengine and channel registers from any interface. However, these registers can be read and written in halt state by executing forced microinstructions, see Section 10.2.7, “Forced Microinstruction Execution.” Immediate data microinstructions may be used to set register values. Some registers are not selectable for immediate data destination, so intermediary register(s), notably P, may have to be used to carry the desired new value to the target register in two or more microinstructions. To preserve original data, any values in the intermediary register(s) must be saved before using the intermediary registers to transfer new values to a final destination. The original, saved values must then be restored after the whole operation.

Similar procedures apply for register reads: their contents must be dumped to SPRAM, where they can be read.

10.2.9 Microengine Flag Access

The microengine halt state allows reading the branch conditions flags through forced microinstructions, but more easily using the dedicated NDEDI register ENGINEx_CFSR. Flag conditions set by the user are visible to the microengine on the next microinstruction execution. The flag set options are limited by the possibilities of forced microinstruction execution.

If the eTPU runs (not single-stepping) after exiting the halted state, the conditions modified during halt may remain only for the first microcycle after the halted state. After the first microcycle, branch conditions are altered only according to their regular update scheme.

10.2.10 Microengine Stall

The microengine can get into a stall state, attending a request from a debug interface signal assertion. The reason for a stall request should be a temporary lack of resources, for instance queue full. During stall the microengine stops execution, but all the other engine logic continues operating: time bases, angle logic, channel logic, input sampling and filters. Stall differs from halt; stall does not enable any of the debug features that halt enables, see Section 10.2.2, “Microengine Halt State.” It also does not break an atomic microengine access, unlike halt.

The microengine can be stalled from the moment TST ends, before executing the first thread microinstruction, until just before the last thread microinstruction is executed. Stall requests are ignored on all other occasions. microengines in a dual engine system can be independently stalled.

10.2.11 SCM Emulation

All SCM implementations are external to the eTPU block. eTPU provides a signal to enable the switching between external SCM banks. The conditions for this switching are:

1. Both engines stopped.
2. VIS bit = 0.

Note that these conditions also stop the clocks of the SCM interface and MISC logic.

10.3 Test Support Features

10.3.1 SCM Test for MISC (Multiple Input Signature Calculator)

The multiple input signature calculator (MISC) comprises special hardware that sequentially reads all SCM positions and calculates, in parallel, a 32-bit signature from a 32-input CRC signature calculator with the following polynomial:

$1 + x^1 + x^2 + x^{22} + x^{31}$

A complete description of the signature calculation procedure can be found in Appendix D, “eTPU MISC Algorithm.”

Once started by the host the MISC runs continuously, restarting after the completion of each cycle. MISC accesses to the SCM array are executed if none of the engines is accessing the SCM, i.e accesses happen while no channel is being serviced to avoid degradation of the microengine performance. An ongoing MISC operation can be aborted by writing 0 to SCMMISEN.

The host must load the register ETPUMISCCMPR, see Section 4.2.3, “eTPU MISC Compare Register (ETPUMISCCMPR),” with the expected value to be found at the end of the MISC cycle, and then start the signature calculation writing bit SCMMISEN=1 in register ETPUMCR, see Section 4.2.1, “eTPU Module Configuration Register (ETPUMCR).” MISC zeroes the signature accumulator and starts reading SCM data and calculating the signature. After last SCM position is read, MISC compares the value in signature accumulator against the value in ETPUMISCCMPR: if there is a mismatch MISC stops, a global exception is issued and the bit SCMMISF in register ETPUMCR assumes value 1. If no mismatch is found, MISC repeats the procedure automatically. When signature is being calculated, SCM address starts at the last SCM address and counts down to 0. The conditions for executing a MISC operation are:

- Both microengines in idle state (no channel is being serviced) or stopped, in any combination (e.g., engine 1 idle with engine 2 stopped).
- ETPUMCR[VIS] = 0.
- ETPUMCR[SCMMISEN] = 1.

If SCMMISEN=0 or VIS=1, the MISC logic stays at its initial state, with address counter pointing to the last SCM position and accumulator reset.

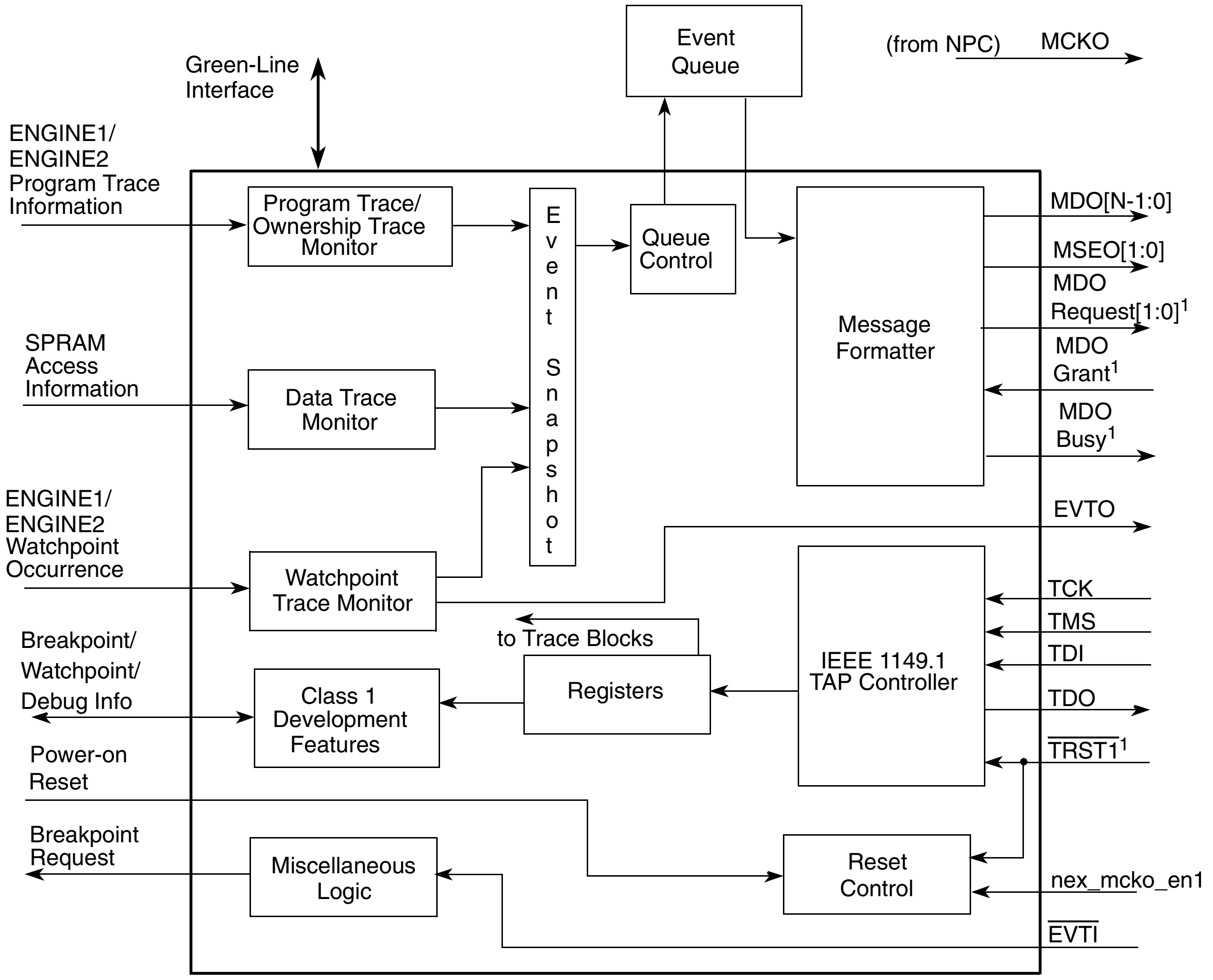
Chapter 11
Nexus Dual eTPU Development Interface (NDEDI)

11.1 Introduction

The Enhanced Timing Processor Unit (eTPU) has its own Nexus class 3 interface, the Nexus Dual eTPU Development Interface (NDEDI). The two eTPU engines and a coherent dual parameter controller (CDC) appear as three separate Nexus clients. The CDC allows the internal eTPU “DMA” accesses to be traced.

11.1.1 Block Diagram

Figure 11-1 is a block diagram of the NDEDI.



1. These signals are used for inter-module communication and are not pins on the MCU.

Figure 11-1. Nexus eTPU Development Interface Block Diagram

11.1.2 Overview

The Nexus eTPU Development Interface (NDEDI) provides real-time development capabilities for the eTPU system including two engines and the coherent dual-parameter controller (CDC) in compliance with the IEEE-ISTO 5001-2002 standard. The NDEDI interfaces to the eTPU via dedicated trace busses. The main development features supported are breakpoint/watchpoint configuration, debug mode register access, enter debug mode at reset negation or during normal execution, single stepping of instructions, branch trace, data trace, ownership trace, and watchpoint trace. Combined, these features make the interface for each engine compliant with Class 3 of the IEEE-ISTO 5001-2002 standard.

Both engines have their own Nexus register sets that allows trace to be set up independently for each of them. The only exception to this is the data trace address range registers that are shared. All the registers are accessed via the JTAG port. Output messages between the engines and other sources are differentiated by the value of the SRC packet in the output messages.

The NDEDI block is combined with similar blocks for other intelligent peripherals, a read/write access block, and a Nexus Port Controller block to provide the complete IEEE-ISTO 5001-2002 interface for a system-on-a-chip design. The JTAG port and Nexus auxiliary port pins are shared by all of the Nexus based blocks on the chip.

11.1.3 Features

The NDEDI block is compliant with the IEEE-ISTO 5001-2002 standard, and implements the following features:

- Full duplex pin interface for medium and high visibility throughput
  - — Pin interface shared among all Nexus-based development interfaces on the device
  - — Ability to select one of two modes during reset: full-port mode (FPM) or reduced-port mode (RPM)
  - — Auxiliary output port
    - – One MCKO (message clock out) pin
    - – MDO (message data out) port
    - – Two $\overline{\text{MSEO}}$ (message start/end out) pins
    - – One $\overline{\text{EVTO}}$ (event out) pin
  - — Auxiliary input port
    - – One $\overline{\text{EVTI}}$ (event in) pin
  - — IEEE 1149.1 (JTAG) Test Access Port (TAP)
    - – Support for optional multi-JTAG TAP Linking Module (TLM)
    - – Four pins (TDI, TDO, TMS, and TCK)
    - – Reset input $\overline{\text{TRST}}$ driven by either the Nexus Port Controller or an external pin
- eTPU development support
  - — IEEE-ISTO 5001-2002 standard class 3 compliant for both engines
  - — Read/write registers in debug mode
  - — Ability to enter debug mode at reset negation or during normal execution
  - — Ability to single step an instruction and re-enter debug mode
  - — Breakpoint and watchpoint configuration
  - — Execute external microcode instructions in debug mode providing indirect read/write accesses to eTPU registers

- Read the microprogram counter in debug mode
- Change the microprogram counter by executing external microcode instructions in debug mode
- Data trace via data write messaging (DWM) and data read messaging (DRM). This allows the development tool to trace reads and writes to selected shared parameter RAM (SPRAM) address ranges.
  - Four data trace windows with programmable address ranges and access attributes are provided. Data trace windowing reduces the requirement on the auxiliary port bandwidth by constraining the number of trace locations. The four trace window address ranges are shared among the dual engines and the eTPU coherent dual-parameter controller (CDC).
- Ownership trace via ownership trace messaging (OTM). OTM provides visibility of which channel is being serviced. An ownership trace message is transmitted to indicate when a new channel service request is scheduled, allowing the development tools to trace task flow. A special OTM is sent when the engine enters in idle, meaning that all requests were serviced and no new requests are yet scheduled.
- Program trace via branch trace messaging (BTM). BTM displays program flow discontinuities (start, jump, return, etc), allowing the development tool to interpolate what transpires between the discontinuities. Thus static code may be traced.
  - Public messages are used for jump, and return instructions.
  - Private messages are used to indicate special eTPU cases not covered by public messages. These messages include the writes to the channel register, the start of a channel service and the trace enabling after a watchpoint hit.
  - Branch/predicate history is kept on direct branches (taken and not taken) and predicated instructions (executed or not executed) to minimize the number of messages transmitted.
- Watchpoint messaging (WPM) via the auxiliary port. WPM provides visibility of the occurrence of the eTPUs' watchpoints.
- Event queue interface. The event queue array is located external to the NDEDI block to separate hard and soft IP.
- Read/write access functions are provided on the device level by a block separate from the NDEDI block.

11.1.4 Modes of Operation

The NDEDI block uses the $\overline{\text{TRST}}$ input and the test-logic-reset state as its primary reset signals. The mode of operation is determined by PCR[FPM] and PCR[MCKO_EN][1] bits in the NPC.

11.1.4.1 Reset

The NDEDI block is placed in reset when nex_reset_b is asserted, or the TAP controller state machine is in the test-logic-reset state. Holding TMS high for 5 consecutive rising edges of TCK guarantees entry into the test-logic-reset state regardless of the current TAP controller state. Asserting nex_reset_b results in asynchronous entry into the reset state. The NDEDI is unaffected by other sources of reset. While in reset, the following actions occur:

- The TAP controller is forced into the test-logic-reset state
- The auxiliary output port pins are negated
- The auxiliary output port enable outputs are negated
- The TDI, TMS, and TCK TAP inputs are ignored
- Registers default back to their reset values

To enter in debug mode from reset condition, the DBE bit in DC register must be programmed to enable debug mode. Then ? must be asserted during 8 clock cycles.

11.1.4.2 Disabled-Port Mode

In disabled-port mode, auxiliary output pin port enable signals are negated, thereby disabling message transmission. Thus, the NDEDI class 3 features are not available in this mode. The primary features available are class 1 features and read/write access to the registers.

11.1.4.3 Full-Port Mode

Full-port mode (FPM) is determined by the PCR[FPM] bit in the NPC block. In full-port mode, the block is enabled and the FPM port enable is asserted indicating that all available MDO pins are to be used for message transmission. All trace features are enabled or can be enabled by writing the configuration registers via the TAP. The number of MDO pins available is 12. Refer to Section 28.2.4, “Modes of Operation,” on page 28-4 for more information.

11.1.4.4 Reduced-Port Mode

Reduced-port mode (RPM) is determined by the PCR[FPM] bit in the NPC block. In reduced-port mode, the block is enabled and the RPM port enable is asserted indicating that only a subset of the MDO pins are to be used for message transmission. All trace features are enabled or can be enabled by writing the configuration registers via the TAP. The number of MDO pins is 4. Refer to Section 28.2.4, “Modes of Operation,” on page 28-4 for more information.

[1]nex_fpm and nex_mcko should be changed only once after exiting test-logic-reset state. Changing this signals in normal operation may lead to unpredictable results.

11.2 Memory Map/Register Definition

This section provides a detailed description of all NDEDI registers accessible to the development tool. Individual bit-level descriptions and reset states of each register are included.

Table 11-1 shows the NDEDI registers by client select and index values. These registers are not memory-mapped and can only be accessed via the TAP.

Table 11-1. NDEDI Memory Map

Index	Register Name	Register Description	CS
1	CSC	Client select control register	X
2	NDEDI_ENGINE1_DC	ENGINE1 development control register	ENGINE1_SRC
4	NDEDI_ENGINE1_DS	ENGINE1 development status register	ENGINE1_SRC
11	NDEDI_ENGINE1_WT	ENGINE1 watchpoint trigger register	ENGINE1_SRC
13	NDEDI_ENGINE1_DTC	ENGINE1 data trace control register	ENGINE1_SRC
22	NDEDI_ENGINE1_BWC1	ENGINE1 breakpoint/watchpoint control register 1	ENGINE1_SRC
23	NDEDI_ENGINE1_BWC2	ENGINE1 breakpoint/watchpoint control register 2	ENGINE1_SRC
24	NDEDI_ENGINE1_BWC3	ENGINE1 breakpoint/watchpoint control register 3	ENGINE1_SRC
30	NDEDI_ENGINE1_BWA1	ENGINE1 breakpoint/watchpoint address register 1	ENGINE1_SRC
31	NDEDI_ENGINE1_BWA2	ENGINE1 breakpoint/watchpoint address register 2	ENGINE1_SRC
38	NDEDI_ENGINE1_BWD1	ENGINE1 breakpoint/watchpoint data register 1	ENGINE1_SRC
39	NDEDI_ENGINE1_BWD2	ENGINE1 breakpoint/watchpoint data register 2	ENGINE1_SRC
64	NDEDI_ENGINE1_PTCE	ENGINE1 program trace channel enable register	ENGINE1_SRC
69	NDEDI_ENGINE1_INST	ENGINE1 microinstruction debug register	ENGINE1_SRC
70	NDEDI_ENGINE1_MPC	ENGINE1 microinstruction program counter	ENGINE1_SRC
71	NDEDI_ENGINE2_CSFR	ENGINE1 channel flag status register	ENGINE1_SRC
2	NDEDI_ENGINE2_DC	ENGINE2 development control register	ENGINE2_SRC
4	NDEDI_ENGINE2_DS	ENGINE2 development status register	ENGINE2_SRC
11	NDEDI_ENGINE2_WT	ENGINE2 watchpoint trigger	ENGINE2_SRC
13	NDEDI_ENGINE2_DTC	ENGINE2 data trace control register	ENGINE2_SRC
22	NDEDI_ENGINE2_BWC1	ENGINE2 breakpoint/watchpoint control register 1	ENGINE2_SRC
23	NDEDI_ENGINE2_BWC2	ENGINE2 breakpoint/watchpoint control register 2	ENGINE2_SRC
24	NDEDI_ENGINE2_BWC3	ENGINE2 breakpoint/watchpoint control register 3	ENGINE2_SRC
30	NDEDI_ENGINE2_BWA1	ENGINE2 breakpoint/watchpoint address register 1	ENGINE2_SRC
31	NDEDI_ENGINE2_BWA2	ENGINE2 breakpoint/watchpoint address register 2	ENGINE2_SRC
38	NDEDI_ENGINE2_BWD1	ENGINE2 breakpoint/watchpoint data register 1	ENGINE2_SRC
39	NDEDI_ENGINE2_BWD2	ENGINE2 breakpoint/watchpoint data register 2	ENGINE2_SRC

Table 11-1. NDEDI Memory Map (continued)

Index	Register Name	Register Description	CS
64	NDEDI_ENGINE2_PTCE	ENGINE2 program trace channel enable register	ENGINE2_SRC
69	NDEDI_ENGINE2_INST	ENGINE2 microinstruction debug register	ENGINE2_SRC
70	NDEDI_ENGINE2_MPC	ENGINE2 microinstruction program counter	ENGINE2_SRC
71	NDEDI_ENGINE2_CSFR	ENGINE2 channel flag status register	ENGINE2_SRC
13	NDEDI_CDC_DTC	CDC data trace control register	CDC_SRC
65	NDEDI_DTAR0	NDEDI data trace address range 0	ENGINE1_SRC or ENGINE2_SRC or CDC_SRC
66	NDEDI_DTAR1	NDEDI data trace address range 1	ENGINE1_SRC or ENGINE2_SRC or CDC_SRC
67	NDEDI_DTAR2	NDEDI data trace address range 2	ENGINE1_SRC or ENGINE2_SRC or CDC_SRC
68	NDEDI_DTAR3	NDEDI data trace address range 3	ENGINE1_SRC or ENGINE2_SRC or CDC_SRC

11.2.1 Register Descriptions

This section consists of NDEDI register descriptions. The ENGINE1 and ENGINE2 registers are shown in Table 11-1. To simplify the document, the description of registers with the same content is only given once instead of being repeated for each ENGINE. This is denoted in the text as ENGINEx where ENGINEx can be replaced by either ENGINE1 or ENGINE2.

11.2.1.1 Client Select Control Register (CSC)

The CSC register is shared by all Nexus blocks on the MCU with registers accessible via the JTAG input port.

	7–K				(K-1)–0			
R	0	0	0	0	CS			
W								
Reset	0	0	0	0	0	0	0	0
Index	1							

Figure 11-2. Client Select Control Register (CSC)

Table 11-2. CSC Field Descriptions

Bits	Name	Description
7–K	—	Reserved.
(K-1)–0	CS	Client select. Determines which client is accessed. The number of bits implemented for the CS field (K) is MCU-dependent, and is the same number of bits as the size of the SRC field transmitted with messages. The value written to the CSC register to access a client register is the same value as the SRC field a client uses in its messages. For example, to access an NDEDI ENGINE1 register, the CSC register is written with a value of ENGINE1_SRC.

11.2.1.2 ENGINE*n* Development Control Register (NDEDI_ENGINE*n*_DC)

The NDEDI_ENGINE*n*_DC register controls various trace features as described below.

When a field is set to a reserved value the behavior defaults to the reset values behavior.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	CHW	PINS	CLKS	CBT	HTWIN	EBC	0	0	0	0	0	0	0	0	0	0
W							▨	CBR	▨	▨	▨	▨	▨	▨	▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	2															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	CBI	DBE	DBR	0	0	0	SS	OVC			EIC		TM		
W	▨				▨	▨	▨									
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	2															

Figure 11-3. ENGINE*n* Development Control Register (NDEDI_ENGINE*n*_DC)

Table 11-3. NDEDI_ENGINE*n*_DC Field Descriptions

Bits	Name	Description
0	CHW	ENGINE*n* CHAN register write trace. Enables the tracing of writes to the CHAN register in the ENGINE*n* execution unit. CHAN register write tracing requires the channel being serviced to have program trace enabled. 0 Tracing not enabled for ENGINE*n* CHAN writes. 1 Tracing enabled for ENGINE*n* CHAN writes
1	PINS	Stop pins in debug mode. Controls whether the ENGINE*n* pins are sampled when the ENGINE*n* enters debug mode. When PINS is set, the pins are not sampled during debug mode or when executing a forced instruction from the microinstruction register. The pins are sampled during normal single steps. 0 Sample pins in the halted state 1 Stop sampling pins in the halted state

Table 11-3. NDEDI_ENGINE*n*_DC Field Descriptions (continued)

Bits	Name	Description
2	CLKS	Stop TCR clocks. Controls whether the TCR clocks from ENGINE*n* stop running when the ENGINE*n* enters debug mode. 0 Do not stop TCRs during debug mode 1 Stop TCRs during debug mode
3	CBT	Client breakpoint timing. Controls the timing of an ENGINE*n* halt due to an external breakpoint source (ipg_debug) or due to a breakpoint at the twin engine. For this bit to matter, CBI or HTWIN must be set to 1. 0 Halt at the end of the current microcycle 1 Halt at the completion of the current instruction thread
4	HTWN	Halt on twin engine breakpoint. Controls, along with CBT, the action taken on breakpoint occurrences at the twin engine.If the twin engine that is causing a breakpoint exit debug state, the ENGINE*n* shall resume its operation. If the HTWIN bit is cleared while the ENGINE*n* is in debug state due to a twin engine breakpoint, the ENGINE*n* shall resume its operation. 0 Do not halt for breakpoints in the twin Engine 1 Halt for twin Engine's breakpoint [1]
5	EBC	$\overline{\text{EVTO}}$ breakpoint controller. Controls the generation of $\overline{\text{EVTO}}$ due to DBR assertion, SW breakpoint, TWIN engine, $\overline{\text{EVTI}}$, MISC error breakpoint, single step breakpoint, external breakpoint (ipg_debug) or system reset (ipg_hard_sync_reset_b). 0 Breakpoint status indication is not output on $\overline{\text{EVTO}}$ 1 Breakpoint status indication is output on $\overline{\text{EVTO}}$
6	—	Reserved.
7	CBR	Clear breakpoint request. Writing this bit to one clears breakpoint requests caused by the following conditions: • ENGINE*n* Internal Breakpoints (data write, CHAN register write, channel service, address match, illegal Instruction) • Software breakpoint • $\overline{\text{EVTI}}$ Breakpoint • MISC Error Breakpoint • Single step breakpoint Asserting this bit causes it to clear one clock later. This means that the development tool can never read this bit with a value other than zero. The eTPU should always execute the first instruction after exiting a breakpoint condition, in such a way that it does not halt twice at the same position.The breakpoints caused by ipg_debug or twin engine are not cleared by the assertion of this bit. 0 No action 1 Clear breakpoint requests
8–16	—	Reserved.
17	CBI	Client breakpoint input. Controls, along with CBT, the action taken on breakpoints from external sources (ipg_debug = 1).The breakpoint condition caused by the external source is only cleared if CBI equals 0 or if the external breakpoint is negated (ipg_debug = 0). 0 Do not halt for other clients' breakpoints 1 Halt for other clients' breakpoints

Table 11-3. NDEDI_ENGINE*n*_DC Field Descriptions (continued)

Bits	Name	Description
18	DBE	Debug enable. Enables debug mode. When DBE is asserted, asserting DBR, system reset, external breakpoint, internal breakpoint occurrences, a halt instruction, a MISC miscompare, or an illegal instruction cause the processor to halt and enter debug mode. [2] This bit is necessary to use features such as single stepping and breakpoints, and does not affect neither watchpoint, program trace nor data trace operations. 0 Debug mode disabled [3] 1 Debug mode enabled [4]
19	DBR	Debug request. Allows the development tool to request processor to enter debug mode. When written, and DBE equals 1 this bit forces the ENGINE*n* to enter debug mode. [5] 0 Exit debug mode if there are no pending breakpoint sources 1 Request debug mode
20–22	—	Reserved.
23	SS	Single step enable. If asserted, a single step occurs when the internal breakpoint conditions are cleared (by asserting CBR or by negating DBR) regardless of the ipg_debug and twin engine breakpoint condition. A single step is also performed if the SS bit is asserted and the engine is in debug mode due to ipg_debug or twin engine. 0 Single step disabled 1 Single step enabled

Table 11-3. NDEDI_ENGINE*n*_DC Field Descriptions (continued)

Bits	Name	Description
24–26	OVC[0:2]	Overrun control. The NDEDI can be programmed to stall the ENGINE*n* operation avoiding errors related to that engine due to full Event Queue. The ENGINE*n* can be stalled in any operations but in the middle of an atomic sequence of accesses to the SPRAM. The worst case happens whenever in Time Slot Transition (TST) [6] where the eTPU can perform up to three atomic accesses to the SPRAM. Since one single microinstruction cycle may generate up to two NDEDI snapshots [7], suppose that the NDEDI asserts a request to delay the engine after the first half of the first access in this specific atomic sequence, then the ENGINE*n* needs to complete the current microinstruction cycle and also needs to perform the entire subsequent micro cycles, thus being able to generate up to five snapshots after the stall request was asserted. The overrun control field controls how the NDEDI reacts when the Queue is about to get full. If programmed to delay the processor, the NDEDI asserts the requests to delay the processor when the Event Queue still has five available positions, thus keeping Event Queue positions to avoid the NDEDI to lose information about the ENGINE*n*,. And negates the delay request as soon as there are six available positions. The request to delay the processor does not affect neither the other engine nor the CDC operations. Therefore, in some extreme cases an overrun message may still be generated even with OVC configured to delay the processor. [8] If an error event happens, the processor will be delayed until the Event Queue is enabled for storing snapshots again, which will happen when the Event Queue gets empty.If the OVC field is changed to 0b011 while the Event Queue has less than 5 available positions the NDEDI will automatically assert the stall request. Thus, ENGINE*n* may still generate an overrun error before entering at the STALL state. If the OVC field is changed to 0b000 while the ENGINE*n* is in STALL state, the NDEDI will automatically negate the stall request. Thus, ENGINE*n* may exit the STALL state. A breakpoint request prevails over the request to delay the processor. Thus, if a breakpoint request happens at the same time of a stall request, the eTPU will enter in HALT state instead of STALL state. 000 Generate overrun message 001–010 Reserved 011 Stall Processor to prevent overruns for ENGINE*n* 100–111 Reserved
27–28	EIC[0:1]	ENGINE*n* $\overline{\text{EVTI}}$ control. The $\overline{\text{EVTI}}$ control (EIC) field can be configured for synchronization or breakpoint generation. If the EIC field is configured to the reserved state, its action reverts to that of the reset state. 00 $\overline{\text{EVTI}}$ assertion causes the next ENGINE*n* program and data trace message to be a synchronization message [9] 01 $\overline{\text{EVTI}}$ assertion causes ENGINE*x* breakpoint generation 10 No operation 11 Reserved
29–31	TM[0:2]	Trace mode.Enables BTM, DTM, and OTM for ENGINE*n*. One or all types of trace may be enabled at a single time by writing to the TM field. NOTE: Trace can also be enabled through watchpoint triggers. The two enable sources are ORed together to form the true trace enable. 000No trace enabled 1xx BTM enabled x1x DTM enabled xx1 OTM enabled

[1]HTWIN causes a breakpoint after the execution of the current microinstruction.

[2]An Illegal instruction causes a breakpoint before the execution of the current microinstruction. A MISC breakpoint causes a breakpoint after the execution of the current microinstruction.

[3]The Engine*n* will resume operation if this bit is cleared while the engine was in a breakpoint.

[4]The HALT instruction within eTPU is ignored if the DBE bit is cleared.

[5]The DBR request cause a breakpoint after the execution of the current microinstruction.

[6]A request to delay the processor while the eTPU is in TST will be recognized after eTPU exits TST and before the first instruction of the thread.

[7]One microcycle of the eTPU corresponds to two system clocks. And since the NDEDI events are evaluated on every single clock, one single microcycle may generate up to two snapshots.

[8]If both registers are programmed to stall while the queue gets full, only the CDC can generate an overrun message.

[9] The next CDC data trace messages are synchronized if the next message from either Engines are synchronized.

11.2.1.3 ENGINEn Development Status Register (NDEDI_ENGINE*n*_DS)

The NDEDI_ENGINE*n*_DS register shows the status of various conditions that impact development support. All status bits are dynamic and do not require clearing. If the Class 3 is enabled, any time the value of any bit in this register changes, a debug status event is queued, and then will generate a debug status message. If the event queue is not enabled to store snapshots while a DS event should be queued, then an error event is queued as soon as the event queue is enabled for storing snapshots.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	HAT	IIBP	CBP	TBP	EBP	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	4															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	BP							0	0	DBS	STP	HWE	HWB	SWB	SSS
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	4															

Figure 11-4. ENGINE*n* Development Status Register (NDEDI_ENGINE*n*_DS)

Table 11-4. NDEDI_ENGINE*n*_DS Field Descriptions

Bits	Name	Description
0–2	—	Reserved.
3	HAT	Halted with active thread. Indicates if the engine was halted during the execution of a thread or while it was IDLE. [1] 0 The eTPU engine is halted with no thread currently active 1 The eTPU engine is halted in the middle of a thread execution
4	IIBP	Illegal instruction breakpoint status. Indicates if there is a breakpoint due to an illegal instruction execution causing the halt condition 0 Breakpoint source is not an illegal instruction execution 1 Breakpoint source is an illegal instruction execution

Table 11-4. NDEDI_ENGINE*n*_DS Field Descriptions (continued)

Bits	Name	Description
5	CBP	Client breakpoint status. Indicates if there is a breakpoint from an external client (ipg_debug) causing the halt condition 0 Breakpoint source is not an external client 1 Breakpoint source is an external client
6	TBP	Twin breakpoint status. Indicates if there is a breakpoint from the twin eTPU engine causing the halt condition 0 Breakpoint source is not the twin eTPU engine 1 Breakpoint source is the twin eTPU engine
7	EBP	$\overline{\text{EVTI}}$ breakpoint status. Indicates if there is a breakpoint caused by $\overline{\text{EVTI}}$ assertion 0 Breakpoint source is not $\overline{\text{EVTI}}$ assertion 1 Breakpoint source is $\overline{\text{EVTI}}$ assertion
8–16	—	Reserved.
17–23	BP	Internal breakpoint status. Shows which hardware breakpoints have occurred causing the ENGINE*n* to enter debug mode. See Table 11-5 for BP bit encodings.
24–25	—	Reserved.
26	DBS	Debug status. Indicates if ENGINE*n* is in debug mode. 0 Processor not halted 1 Processor halted in debug mode
27	STP	Stop status. Indicates if the ENGINE*n* is stopped in low power mode. 0 Processor not stopped 1 Processor stopped in low power mode
28	HWE	Hardware error status. Indicates if the ENGINE*n* has encountered a HW error. The only hardware error defined for the ENGINE is a MISC miscompare. 0 No HW error 1 Non-recoverable HW error occurred
29	HWB	Hardware breakpoint status. Indicates if a hardware breakpoint (e.g address comparator) has occurred. 0 No hardware breakpoint occurred 1 Hardware breakpoint occurred
30	SWB	Software breakpoint status. Indicates if a software breakpoint has occurred. A software breakpoint occurs with the execution of the HALT instruction. 0 No software breakpoint occurred 1 Software breakpoint occurred
31	SSS	Single step status. Indicates if the processor is halted for debug mode due to single step occurrence. 0 Processor not halted due to single step 1 Processor halted in debug mode due to single step

[1]The TST state of the eTPU is considered to be an active state.

Table 11-5. BP Values

Value	Description
0b0000000	No breakpoint condition occurred
0b1xxxxxx	eTPU breakpoint 1 (based on BWC1 register)
0bx1xxxxx	eTPU breakpoint 2 (based on BWC2 register)
0bxx1xxxx	Channel register write breakpoint
0bxxx1xxx	Host service request breakpoint
0bxxxx1xx	Link register breakpoint
0bxxxxx1x	MRL breakpoint
0bxxxxxx1	TDL breakpoint

11.2.1.4 ENGINE*n* Watchpoint Trigger Register (NDEDI_ENGINE*n*_WT)

The NDEDI_ENGINE*n*_WT register allows traces to be enabled and/or disabled on the occurrence of a watchpoint. If the same watchpoint is programed to enable and disable a trace, the occurrence of the watchpoint toggles the current value of the trace flags. If one watchpoint occurs to enable trace and a different watchpoint occurs to disable trace at the same time the current value of the trace flags will also be toggled. Trace enable flags from triggers (PTEF and DTEF) are ORed with the appropriate bit in the TM field of the DC register and sent to the trace blocks.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	PTS			PTE			DTS			DTE			0	0	0	0
W													▨	▨	▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	11															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	PTEF	DTEF
W	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	PTEC	DTEC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	11															

Figure 11-5. ENGINE*n* Watchpoint Trigger Register (NDEDI_ENGINE*n*_WT)

Table 11-6. NDEDI_ENGINE*n*_WT Field Descriptions

Bits	Name	Description
0–2	PTS[0:2]	Program trace start. Allows program trace to be enabled at a watchpoint occurrence. 000 Trigger disabled 001 Program trace enabled on eTPU watchpoint 1 occurrence 010 Program trace enabled on eTPU watchpoint 2 occurrence 011 Program trace enabled on channel register write watchpoint occurrence 100 Program trace enabled on host service request watchpoint occurrence 101 Program trace enabled on link register watchpoint occurrence 111 Program trace enabled on TDL watchpoint occurrence
3–5	PTE[0:2]	Program trace end. Allows program trace to be disabled at a watchpoint occurrence. 000 Trigger disabled 001 Program trace disabled on eTPU Watchpoint 1 occurrence 010 Program trace disabled on eTPU watchpoint 2 occurrence 011 Program trace disabled on channel register write watchpoint occurrence 100 Program trace disabled on host service request watchpoint occurrence 101 Program trace disabled on link register watchpoint occurrence 110 Program trace disabled on MRL watchpoint occurrence 111 Program trace disabled on TDL watchpoint occurrence
6–8	DTS[0:2]	Data trace start. Allows data trace to be enabled at a watchpoint occurrence. 000 Trigger disabled 001 Data trace enabled on eTPU watchpoint 1 occurrence 010 Data trace enabled on eTPU watchpoint 2 occurrence 011 Data trace enabled on channel register write watchpoint occurrence 100 Data trace enabled on host service request watchpoint occurrence 101 Data trace enabled on link register watchpoint occurrence 110 Data trace enabled on MRL watchpoint occurrence 111 Data trace enabled on TDL watchpoint occurrence
9–11	DTE[0:2]	Data trace end. Allows data trace to be disabled at a watchpoint occurrence. 000 Trigger disabled 001 Data trace disabled on eTPU watchpoint 1 occurrence 010 Data trace disabled on eTPU watchpoint 2 occurrence 011 Data trace disabled on channel register write watchpoint occurrence 100 Data trace disabled on host service request watchpoint occurrence 101 Data trace disabled on host service request watchpoint occurrence 110 Data trace disabled on MRL watchpoint occurrence 111 Data trace disabled on TDL watchpoint occurrence
12–29	—	Reserved.
30	PTEF	Program trace enable flag. Shows if program trace is currently enabled due to a watchpoint trigger. 0 Program trace not enabled due to watchpoint trigger 1 Program trace enabled due to watchpoint trigger
30	PTEC	Program trace enable clear. Writing a one to the PTEC bit clears the PTEF flag. This allows the development tool to turn off program trace started via a trigger without using a second trigger or module reset. 0 Keep PTEF flag unaltered 1 Clear the PTEF flag

Table 11-6. NDEDI_ENGINE*n*_WT Field Descriptions (continued)

Bits	Name	Description
31	DTEF	Data trace enable flag. Shows if data trace is currently enabled due to a watchpoint trigger. 0 Data Trace not enabled due to watchpoint trigger 1 Data Trace enabled due to watchpoint trigger
31	DTEC	Data trace enable clear. Writing a one to the DTEC bit clears the DTEF flag. This allows the development tool to turn off data trace started via a trigger without using a second trigger or module reset. 0 Keep DTEF flag unaltered 1 Clear the DTEF flag

11.2.1.5 ENGINE*n* Data Trace Control Register (NDEDI_ENGINE*n*_DTC)

The NDEDI_ENGINE*n*_DTC register controls which address ranges are enabled for data trace, and if reads and/or writes are traced in that range.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	RWT0		RWT1		RWT2		RWT3		0	0	0	0	0	0	0	0
W									▨	▨	▨	▨	▨	▨	▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	13															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	RC0	RC1	RC2	RC3	0	0	0	0
W	▨	▨	▨	▨	▨	▨	▨	▨					▨	▨	▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	13															

Figure 11-6. ENGINE*n* Data Trace Control Register (NDEDI_ENGINE*n*_DTC)

Table 11-7. NDEDI_ENGINE*n*_DTC Field Descriptions

Bits	Name	Description
0–1	RWT0[0:1]	ENGINE*n* read/write trace 0 control. Controls whether data trace messages are generated for ENGINE*n* accesses inside eTPU data trace window 0 (see Section 11.2.1.15, “Data Trace Address Range 0 Register (NDEDI_DTAR0)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No ENGINE*n* data trace messages generated for eTPU window 0 01 Enable ENGINE*n* data read trace for eTPU window 0 10 Enable ENGINE*n* data write trace for eTPU window 0 11 Enable ENGINE*n* data read and write trace for eTPU window 0
2–3	RWT1[0:1]	ENGINE*n* read/write trace 1 control. Controls whether data trace messages are generated for ENGINE*n* accesses inside eTPU data trace window 1 (see Section 11.2.1.16, “Data Trace Address Range 1 Register (NDEDI_DTAR1)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No ENGINE*n* data trace messages generated for eTPU window 1 01 Enable ENGINE*n* data read trace for eTPU window 1 10 Enable ENGINE*n* data write trace for eTPU window 1 11 Enable ENGINE*n* data read and write trace for eTPU window 1
4–5	RWT2[0:1]	ENGINE*n* read/write trace 2 control. Controls whether data trace messages are generated for ENGINE*n* accesses inside eTPU data trace window 2 (see Section 11.2.1.17, “Data Trace Address Range 2 Register (NDEDI_DTAR2)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No ENGINE*n* data trace messages generated for eTPU window 2 01 Enable ENGINE*n* data read trace for eTPU window 2 10 Enable ENGINE*n* data write trace for eTPU window 2 11 Enable ENGINE*n* data read and write trace for eTPU window 2
6–7	RWT3[0:1]	ENGINE*n* read/write trace 3 control. Controls whether data trace messages are generated for ENGINE*n* accesses inside eTPU data trace window 3 (see Section 11.2.1.18, “Data Trace Address Range 3 Register (NDEDI_DTAR3)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No ENGINE*n* data trace messages generated for eTPU window 3 01 Enable ENGINE*n* data read trace for eTPU window 3 10 Enable ENGINE*n* data write trace for eTPU window 3 11 Enable ENGINE*n* data read and write trace for eTPU window 3
8–23	—	Reserved.
24	RC0	Range control 0. Controls which addresses match data trace window 0. 0 Trace address inside (inclusive) of data trace window 0 1 Trace addresses outside (exclusive) of data trace window
25	RC1	Range control 1. Controls which addresses match data trace window 1. 0 Trace address inside (inclusive) of data trace window 1 1 Trace addresses outside (exclusive) of data trace window 1
26	RC2	Range control 2. Controls which addresses match data trace window 2. 0 Trace address inside (inclusive) of data trace window 2 1 Trace addresses outside (exclusive) of data trace window 2
27	RC3	Range control 3. Controls which addresses match data trace window 3. 0 Trace address inside (inclusive) of data trace window 3 1 Trace addresses outside (exclusive) of data trace window 3
28–31	—	Reserved.

11.2.1.6 ENGINE*n* Breakpoint/Watchpoint Control Registers (NDEDI_ENGINE*n*_BWC1, NDEDI_ENGINE*n*_BWC2)

The NDEDI_ENGINE*n*_BWC1 register is used to configure ENGINE*n* breakpoint/watchpoint 1 along with the NDEDI_ENGINE*n*_BWA1 and NDEDI_ENGINE*n*_BWD1 registers.

The NDEDI_ENGINE*n*_BWC2 register is used to configure ENGINE*n* breakpoint/watchpoint 2 along with the NDEDI_ENGINE*n*_BWA2 and NDEDI_ENGINE*n*_BWD2 registers.

When a field is set to a reserved value the behavior defaults to the reset values behavior.

The register can be configured for breakpoints based on data write values, instruction address and data read or write addresses.

When a breakpoint occurs the program execution is halted before the instruction is executed. This means that data access breakpoints occur before the SPRAM access actually takes place.

The data accesses that occur within the time slot transition (TST) should not generate breakpoints. Thus, it is not expected that the eTPU system signalize these transactions to the NDEDI interface.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	BWE		BRW		0	0	0	0	BME				BSU		BWO	
W					▨	▨	▨	▨					▨	▨		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	22															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	BWO	EOC	0	0	0	SCMSK					0	SCMV				
W			▨	▨	▨						▨					
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	22 (NDEDI_ENGINE*n*_BWC1); 23 (NDEDI_ENGINE*n*_BWC2)															

Figure 11-7. ENGINE*n* Breakpoint/Watchpoint Control Registers (NDEDI_ENGINE*n*_BWC1, NDEDI_ENGINE*n*_BWC2)

Table 11-8. NDEDI_ENGINE*n*_BWC*n* Field Descriptions

Bits	Name	Description
0–1	BWE[0:1]	Breakpoint/watchpoint enable. Enables the ENGINE*n* breakpoint or watchpoint. 00 Disabled 01 Breakpoint enabled 10 Reserved 11 Watchpoint enabled
2–3	BRW[0:1]	Breakpoint/watchpoint read/write select. Selects whether read and/or write accesses cause a breakpoint/watchpoint. 00 Match on read SPRAM access 01 Match on write SPRAM access 10 Match on any SPRAM accesses 11 Reserved
4–7	—	Reserved.
8–11	BME[0:3]	Breakpoint/watchpoint data mask enable. Selects which data bytes are used for breakpoint/watchpoint generation. 0000 All bytes of data compared 1xxx Most significant byte of data masked out xxx1 Least significant byte of data masked out
12–13	BSU[0:1]	Breakpoint/watchpoint data size unit. Indicates the data size unit used by the NDEDI block. This field always reads as 0b00 to indicate the data size unit is 1 byte.
14–16	BWO[0:2]	Breakpoint/watchpoint operand.Selects whether address and/or data matching is done and if matching is done on data fetches or instruction fetches. 00x Breakpoint/Watchpoint Disabled x10 Reserved 011 Compare data of a SPRAM write access with BWD 100 Compare instruction Address with BWA 101 Compare address of a SPRAM write/read access with BWA 111 Compare data and address of a SPRAM write access with BWD and BWA respectively [1]
17	EOC	$\overline{\text{EVTO}}$ control. Controls the action of the $\overline{\text{EVTO}}$ output at the occurrence of a breakpoint or watchpoint. 0 Breakpoint/watchpoint status indication is not output on $\overline{\text{EVTO}}$ 1 Breakpoint/watchpoint status indication is output on $\overline{\text{EVTO}}$
18–20	—	Reserved.
21–25	SCMSK[0:4]	Serviced channel mask value. Used to mask what bits of the SCMV field are compared to the serviced channel. When a bit in the SCMSK field is 0, that bit is not compared for masking purposes. Serviced channel match = (SCMV&SCMSK) == (Serviced Channel & SCMSK);
26	—	Reserved.
27–31	SCMV[0:5]	Serviced channel match value.The SCMV field value is compared against the ENGINE*n* serviced channel when generating the breakpoints and watchpoints.

[1] A breakpoint/watchpoint occurrence only happens if there is a match in both BWD and BWA.

11.2.1.7 ENGINE*n* Breakpoint/Watchpoint Address Registers (NDEDI_ENGINE*n*_BWA1, NDEDI_ENGINE*n*_BWA2)

The NDEDI_ENGINE*n*_BWA1 register is used to configure ENGINE*n* breakpoint/watchpoint 1 along with the NDEDI_ENGINE*n*_BWC1 and NDEDI_ENGINE*n*_BWD1 registers. It is used for data access and instruction fetches breakpoints.

The NDEDI_ENGINE*n*_BWA2 register is used to configure ENGINE*n* breakpoint/watchpoint 2 along with the NDEDI_ENGINE*n*_BWC2 and NDEDI_ENGINE*n*_BWD2 registers. It is used for data access and instruction fetches breakpoints.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	BWAM														0	0
W															▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	30 (NDEDI_ENGINEx_BWA1); 31(NDEDI_ENGINEx_BWA2)															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	BWA														0	0
W															▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	30 (NDEDI_ENGINEx_BWA1); 31(NDEDI_ENGINEx_BWA2)															

Figure 11-8. ENGINE*n* Breakpoint/Watchpoint Address Registers (NDEDI_ENGINE*n*_BWA1, NDEDI_ENGINE*n*_BWA2)

Table 11-9. NDEDI_ENGINE*n*_BWA*n* Field Descriptions

Bits	Name	Description
0–13	BWAM[0:13]	Breakpoint/watchpoint address mask. Used to select which address bits are compared for breakpoint and watchpoint generation. Address match = (BWA&BWAM) == (Address & BWAM)
14–15	—	Reserved.
16–29	BWA[0:13]	Breakpoint/watchpoint address. Used to compare address operands (address of instruction or data). The BWA field represent a word addressable address. For SPRAM accesses, any access inside that word matches. The most significant two bits are ignored for address of data comparisons because the SPRAM address space is only 14 bits.
30–31	—	Reserved.

11.2.1.8 ENGINEx Breakpoint/Watchpoint Data Registers (NDEDI_ENGINE*n*_BWD1, NDEDI_ENGINE*n*_BWD2)

The NDEDI_ENGINE*n*_BWD1 register is used to configure ENGINE*n* breakpoint/watchpoint 1 along with the NDEDI_ENGINE*n*_BWC1 and NDEDI_ENGINE*n*_BWA1 registers. It is used for data access breakpoints only.

The NDEDI_ENGINE*n*_BWD2 register is used to configure ENGINE*n* breakpoint/watchpoint 2 along with the NDEDI_ENGINE*n*_BWC2 and NDEDI_ENGINE*n*_BWA2 registers. It is used for data access breakpoints only.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	BWD															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	38 (NDEDI_ENGINE*n*_BWD1); 39 (NDEDI_ENGINE*n*_BWD2)															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	BWD															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	38 (NDEDI_ENGINE*n*_BWD1); 39 (NDEDI_ENGINE*n*_BWD2)															

Figure 11-9. ENGINE*n* Breakpoint/Watchpoint Data Registers (NDEDI_ENGINE*n*_BWD1, NDEDI_ENGINE*n*_BWD2)

Table 11-10. NDEDI_ENGINE*n*_BWD*n* Field Descriptions

Bits	Name	Description
0–31	BWD[0:31]	Breakpoint/watchpoint data. Used to compare data operands whether in a SPRAM write access. Depending on the size of the access, only certain bits of BWD are compared against the data written to the SPRAM. The table below shows which bits of BWD are compared against the data written to the SPRAM.

Operation Size	Bits Compared
8-bits	8 most significant bits of BWD
24-bits	24 least significant bits of BWD
32-bits	All bits from BWD

11.2.1.9 ENGINE*n* Program Trace Channel Enable Register (NDEDI_ENGINE*n*_PTCE)

The NDEDI_ENGINE*n*_PTCE register enables program tracing for each of the 32 ENGINE*n* channels. There is one enable bit per channel. And the bit corresponding to the serviced channel is the only bit tested for enabling program trace. For these bits to have any effect, program trace must also be enabled in the TM field of the DC register or via a watchpoint trigger.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	ENGINE*n*_ PTCE31	ENGINE*n*_ PTCE30	ENGINE*n*_ PTCE29	ENGINE*n*_ PTCE28	ENGINE*n*_ PTCE27	ENGINE*n*_ PTCE26	ENGINE*n*_ PTCE25	ENGINE*n*_ PTCE24	ENGINE*n*_ PTCE23	ENGINE*n*_ PTCE22	ENGINE*n*_ PTCE21	ENGINE*n*_ PTCE20	ENGINE*n*_ PTCE19	ENGINE*n*_ PTCE18	ENGINE*n*_ PTCE17	ENGINE*n*_ PTCE16
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Index	64															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	ENGINE*n*_ PTCE15	ENGINE*n*_ PTCE14	ENGINE*n*_ PTCE13	ENGINE*n*_ PTCE12	ENGINE*n*_ PTCE11	ENGINE*n*_ PTCE10	ENGINE*n*_ PTCE9	ENGINE*n*_ PTCE8	ENGINE*n*_ PTCE7	ENGINE*n*_ PTCE6	ENGINE*n*_ PTCE5	ENGINE*n*_ PTCE4	ENGINE*n*_ PTCE3	ENGINE*n*_ PTCE2	ENGINE*n*_ PTCE1	ENGINE*n*_ PTCE0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Index	64															

Figure 11-10. ENGINE*n* Program Trace Channel Enable Register (NDEDI_ENGINE*n*_PTCE)

Table 11-11. NDEDI_ENGINE*n*_PTCE Field Descriptions

Bits	Name	Description
0–31	ENGINE*n*_PTCE*n*	ENGINE*n* timer channel *n* program trace enable. 0 Program tracing disabled for ENGINE*n* timer channel *n* 1 Program tracing enabled for ENGINE*n* timer channel *n*

11.2.1.10 ENGINE*n* Breakpoint/Watchpoint Control 3 Register (NDEDI_ENGINE*n*_BWC3)

The NDEDI_ENGINE*n*_BWC3 register is used to configure miscellaneous ENGINE*n* breakpoint/watchpoint sources. This is a vendor-defined register with different fields that the other breakpoint/watchpoint control registers.

When a field is set to a reserved value the behavior defaults to the reset values behavior.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	SCMV					0	0	0	SCMSK				
W	▨	▨	▨						▨	▨	▨					
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	24															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	BCRW			BHSR			BLINK			BMRL			BTDL		
W	▨															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	24															

Figure 11-11. ENGINE*n* Breakpoint/Watchpoint Control 3 Register (NDEDI_ENGINE*n*_BWC3)

Table 11-12. NDEDI_ENGINE*n*_BWC3 Field Descriptions

Bits	Name	Description
0–2	—	Reserved.
3–7	SCMV[0:4]	Serviced channel match value. Compared against the serviced channel register when generating a breakpoint due to a write in the CHAN register or due to the start of a service.
8–10	—	Reserved.
11–15	SCMSK[0:4]	Serviced channel mask value. Used to mask what bits of the SCMV field are compared to the serviced channel register when generating a breakpoint due to a write in the CHAN register or due to the start of a service. When a bit in the SCMSK field is 0, that bit is not compared for masking purposes.
16	—	Reserved.
17–19	BCRW[0:2]	Break on channel register write. Configures the action when there is a write to the CHAN register and there is a serviced channel match (as described below). The breakpoint causes execution to halt at the completion of the current microinstruction, which means after the CHAN register is written. Serviced channel match = (SCMV&SCMSK) == (Serviced Channel & SCMSK) 00x Disabled 01x Breakpoint enabled 10x Reserved 11x Watchpoint enabled xx0 Breakpoint/watchpoint status indication not output on $\overline{\text{EVTO}}$ xx1 Breakpoint/watchpoint status indication output on $\overline{\text{EVTO}}$

Table 11-12. NDEDI_ENGINE*n*_BWC3 Field Descriptions (continued)

Bits	Name	Description
20–22	BHSR[0:2]	Break on host service request. Configures the action when there is a serviced channel match (as described below) and the host service request is asserted at the beginning of the channel service. The breakpoint causes execution to halt after the time slot transition (TST) completes but before the first instruction of the thread is executed. Serviced channel match = (SCMV&SCMSK) == (Serviced Channel & SCMSK) 00x Disabled 01x Breakpoint enabled 10x Reserved 11x Watchpoint enabled xx0 Breakpoint/watchpoint status indication not output on $\overline{\text{EVTO}}$ xx1 Breakpoint/watchpoint status indication output on $\overline{\text{EVTO}}$
23–25	BLINK[0:2]	Break on link service request. Configures the action when there is a serviced channel match (as described below) and the link service register is asserted at the beginning of the channel service. The breakpoint causes execution to halt after the time slot transition (TST) completes but before the first instruction of the thread is executed. Serviced channel match = (SCMV&SCMSK) == (Serviced Channel & SCMSK) 00x Disabled 01x Breakpoint enabled 10x Reserved 11x Watchpoint enabled xx0 Breakpoint/watchpoint status indication not output on $\overline{\text{EVTO}}$ xx1 Breakpoint/watchpoint status indication output on $\overline{\text{EVTO}}$
26–28	BMRL[0:2]	Break on match recognition request. Configures the action when there is a serviced channel match (as described below) and either MRL1 or MRL_B is asserted at the beginning of the channel service. The breakpoint causes execution to halt after the time slot transition (TST) completes but before the first instruction of the thread is executed. Serviced channel match = (SCMV&SCMSK) == (Serviced Channel & SCMSK) 00x Disabled 01x Breakpoint enabled 10x Reserved 11x Watchpoint enabled xx0 Breakpoint/watchpoint status indication not output on $\overline{\text{EVTO}}$ xxx1 Breakpoint/watchpoint status indication output on $\overline{\text{EVTO}}$
29–31	BTDL[0:2]	Break on transition detect request.Configures the action when there is a serviced channel match (as described below) and the TDL_A or TDL_B is asserted at the beginning of the channel service. The breakpoint causes execution to halt after the time slot transition (TST) completes but before the first instruction of the thread is executed. Serviced channel match = (SCMV&SCMSK) == (Serviced Channel & SCMSK) 00x Disabled 01x Breakpoint enabled 10x Reserved 11x Watchpoint enabled xx0 Breakpoint/watchpoint status indication not output on $\overline{\text{EVTO}}$ xx1 Breakpoint/watchpoint status indication output on $\overline{\text{EVTO}}$

11.2.1.11 ENGINE*n* Microinstruction Debug Register (NDEDI_ENGINE*n*_INST)

The NDEDI_ENGINE*n*_INST register is used for forcing a microinstruction on the microinstruction register. This register should only be accessed while the ENGINE*n* is halted. If accessed while the ENGINE*n* is not halted, writes are ignored.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	ENGINE*n*_INST															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	69															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	ENGINE*n*_INST															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	69															

Figure 11-12. ENGINE*n* Microinstruction Debug Register (NDEDI_ENGINE*n*_INST)

Writes to the register in debug mode force the execution of an arbitrary instruction. This is used to retrieve information about internal status, set internal flag values among other things during debug mode. Some fields of this microinstruction may be ignored, for instance an END instruction will be ignored. For further information about the arbitrary instruction refer to the.

11.2.1.12 ENGINE*n* Microprogram Counter Debug Register (NDEDI_ENGINE*n*_MPC)

The NDEDI_ENGINE*n*_MPC register is used for reading the microprogram counter value. This register is only meaningful while the ENGINE*n* is halted. If accessed while the ENGINE*n* is not halted, reads return zero. Writes to this register are always ignored.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	70															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	NDEDI_ENGINE*n*_MPC															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	70															

Figure 11-13. ENGINE*n* Microprogram Counter Debug Register (NDEDI_ENGINE*n*_MPC)

11.2.1.13 ENGINE*n* Channel Flag Status Register (NDEDI_ENGINE*n*_CFSR)

The NDEDI_ENGINE*n*_CFSR register provides read-only access to the flags of the channel being serviced by ENGINE*n*. Reads to this register when ENGINE*n* is out of halt mode return zero. Transition detection and match recognition flags are available in two versions: one reflecting the branch condition selection (BCC), TDL_A, TDL_B, MRL_A, and MRL_B, and an ‘internal’ version, ITDL1, ITDL2, IMR1, and IMR2, reflecting the flag value within the channel logic.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	V	N	C	Z	MV	MN	MC	MZ	TDL_A	TDL_B	MRL_A	MRL_B	LSR	SMLCK	FM	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	71															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	SRI	HSR			FLAG0	FLAG1	MRLE1	MRLE2	ITDL1	ITDL2	IMRL1	IMRL2	PSS	PSTI	PSTO	OBE
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	71															

Figure 11-14. ENGINE*n* Channel Flag Status Register (NDEDI_ENGINE*n*_CSFR)

Table 11-13. NDEDI_ENGINE*n*_CSFR Field Descriptions

Bits	Name	Description
0	V	EAU overflow flag. Overflow status flag from an EAU operation. 0 EAU overflow flag is cleared 1 EAU overflow flag is set
1	N	EAU negative flag. Negative status flag from an EAU operation. 0 EAU negative flag is cleared 1 EAU negative flag is set
2	C	EAU carry flag. Carry status flag from an EAU operation. 0 EAU carry flag is cleared 1 EAU carry flag is set
3	Z	EAU carry flag. Zero status flag from an EAU operation. 0 EAU zero flag is cleared 1 EAU zero flag is set
4	MV	MAC/divide unit overflow flag. Overflow status flag from an MAC/Divide Unit operation. 0 MAC/divide unit overflow flag is cleared 1 MAC/divide unit overflow flag is set
5	MN	MAC/divide unit negative flag. Negative status flag from an MAC/Divide Unit operation. 0 MAC/Divide Unit negative flag is cleared 1 MAC/Divide Unit negative flag is set
6	MC	MAC/divide unit carry flag. Carry status flag from an MAC/Divide Unit operation. 0 MAC/Divide Unit carry flag is cleared 1 MAC/Divide Unit carry flag is set
7	MZ	MAC/Divide unit zero flag. Zero status flag from an MAC/Divide Unit operation. 0 MAC/Divide Unit zero flag is cleared 1 MAC/Divide Unit zero flag is set
8	TDL_A	Channel transition detection latch 1. Channel transition detection status flag 1 as available in the BCC. 0 Channel Transition flag 1 in BCC is cleared 1 Channel Transition flag 1 in BCC is set
9	TDL_B	Channel transition detection latch 2. Channel transition detection status flag 2 as available in the BCC. 0 Channel Transition flag 2 in BCC is cleared 1 Channel Transition flag 2 in BCC is set
10	MRL1	Channel match recognition latch 1. Channel Match Recognition status flag 1 as available in the BCC. 0 Match Recognition flag 1 in BCC is cleared 1 Match Recognition flag 1 in BCC is set
11	MRL2	Channel match recognition latch 2. Channel Match Recognition status flag 2 as available in the BCC. 0 Match Recognition flag 2 in BCC is cleared 1 Match Recognition flag 2 in BCC is set
12	LSR	Channel link service register. Channel link service register flag. 0 Link even to currently serviced channel did not occur 1 Link event to currently serviced channel occurred
13	SMLCK	Channel semaphore flag. 0 Semaphore flag is cleared 1 Semaphore flag is set

Table 11-13. NDEDI_ENGINE*n*_CSFR Field Descriptions (continued)

Bits	Name	Description
14–15	FM[0:1]	Channel function mode. Reflects the state of the FM field of the status control register of the current channel.
16	SRI	Service request inhibit. Flag that blocks channel service requests due to the assertion of MRL1/2 and/or TDL_A/B. 0 Service requests not inhibited 1 Service requests inhibited
17–19	HSR[0:2]	Host service request. Indicates the entry point HSR of the thread currently executing, if any. This was the value of HSR of the serviced channel when the entry point was chosen. It is not necessarily the value of the host service request register of the current channel.
20	FLAG0	Channel state resolution flag 0. One of two flags per channel that can be set or cleared by microcode to further resolve the entry point for a channel service. 0 FLAG0 was cleared by microcode 1 FLAG0 was set by microcode
21	FLAG1	Channel state resolution flag 1. One of two flags per channel that can be set or cleared by microcode to further resolve the entry point for a channel service. 0 FLAG1 was cleared by microcode 1 FLAG1 was set by microcode
22	MRLE1	Channel match recognition latch enable 1. 0 Match recognition disabled for event 1 1 Match recognition enabled for event 1
23	MRLE2	Channel match recognition latch enable 2. 0 Match recognition disabled for event 2 1 Match recognition enabled for event 2
24	ITDL1	Internal channel transition detection latch 1. Channel transition detection status flag 1 reflecting its status directly in the channel logic. 0 Channel transition flag 1 within the channel logic is cleared 1 Channel transition flag 1 within the channel logic is set
25	ITDL2	Internal channel transition detection latch 2. Channel transition detection status flag 2 reflecting its status directly in the channel logic. 0 Channel transition flag 2 within the channel logic is cleared 1 Channel transition flag 2 within the channel logic is set
26	IMRL1	Internal channel match recognition latch 1. Channel match recognition status flag 1 reflecting its status directly in the channel logic. 0 Match recognition flag 1 within the channel logic is cleared 1 Match recognition flag 1 within the channel logic is set
27	IMRL2	Internal channel match recognition latch 2. Channel match recognition status flag 2 reflecting its status directly in the channel logic. 0 Match recognition flag 2 within the channel logic is cleared 1 Match recognition flag 2 within the channel logic is set
28	PSS	Pin sampled state. PReflects the status of the PSS flag for the current channel. 0 PSS is cleared 1 PSS is set
29	PSTI	Pin state input. Reflects the status of the filtered input signal for the current channel. 0 Filtered input signal is cleared 1 Filtered input signal is set

Table 11-13. NDEDI_ENGINE*n*_CSFR Field Descriptions (continued)

Bits	Name	Description
30	PSTO	Pin state output. Reflects the status of the value driven by the output control logic of the current channel. 0 Output pin state register is cleared 1 Output pin state register is set
31	OBE	Output buffer enable. Reflects the status of the OBE control register for the current channel. 0 Output buffer enable is cleared 1 Output buffer enable is set

11.2.1.14 CDC Data Trace Control Register (NDEDI_CDC_DTC)

The NDEDI_CDC_DTC register controls which address ranges are enabled for CDC access data trace, and if reads and/or writes are traced in that range.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	RWT0		RWT1		RWT2		RWT3		0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	13															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	RC0	RC1	RC2	RC3	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	13															

Figure 11-15. CDC Data Trace Control Register (NDEDI_CDC_DTC)

Table 11-14. NDEDI_CDC_DTC Field Descriptions

Bits	Name	Description
0–1	RWT0[0:1]	CDC read/write trace 0 control. Controls whether data trace messages are generated for CDC accesses inside eTPU data trace window 0 (see Section 11.2.1.15, “Data Trace Address Range 0 Register (NDEDI_DTAR0)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No CDC data trace messages generated for eTPU window 0 01 Enable CDC data read trace for eTPU window 0 10 Enable CDC data write trace for eTPU window 0 11 Enable CDC data read and write trace for eTPU window 0
2–3	RWT1[0:1]	CDC read/write trace 1 control. Controls whether data trace messages are generated for CDC accesses inside eTPU data trace window 1 (see Section 11.2.1.16, “Data Trace Address Range 1 Register (NDEDI_DTAR1)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No CDC data trace messages generated for eTPU window 1 01 Enable CDC data read trace for eTPU window 1 10 Enable CDC data write trace for eTPU window 1 11 Enable CDC data read and write trace for eTPU window 1
4–5	RWT2[0:1]	CDC read/write trace 2 control. Controls whether data trace messages are generated for CDC accesses inside eTPU data trace window 2 (see Section 11.2.1.17, “Data Trace Address Range 2 Register (NDEDI_DTAR2)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No CDC data trace messages generated for eTPU window 2 01 Enable CDC data read trace for eTPU window 2 10 Enable CDC data write trace for eTPU window 2 11 Enable CDC data read and write trace for eTPU window 2
6–7	RWT3[0:1]	CDC read/write trace 3 control. Controls whether data trace messages are generated for CDC accesses inside eTPU data trace window 3 (see Section 11.2.1.18, “Data Trace Address Range 3 Register (NDEDI_DTAR3)”), and if so, whether reads, writes, or both generate the data trace messages. 00 No CDC data trace messages generated for eTPU window 3 01 Enable CDC data read trace for eTPU window 3 10 Enable CDC data write trace for eTPU window 3 11 Enable CDC data read and write trace for eTPU window 3
8–23	—	Reserved.
24	RC0	Range control 0. Controls which addresses match data trace window 0. 0 Trace address inside (inclusive) of data trace window 0 1 Trace addresses outside (exclusive) of data trace window 0
25	RC1	Range control 1. Controls which addresses match data trace window 1. 0 Trace address inside (inclusive) of data trace window 1 1 Trace addresses outside (exclusive) of data trace window 1
26	RC2	Range control 2. Controls which addresses match data trace window 2. 0 Trace address inside (inclusive) of data trace window 2 1 Trace addresses outside (exclusive) of data trace window 2
27	RC3	Range control 3. Controls which addresses match data trace window 3. 0 Trace address inside (inclusive) of data trace window 3 1 Trace addresses outside (exclusive) of data trace window 3
28–31	—	Reserved.

11.2.1.15 Data Trace Address Range 0 Register (NDEDI_DTAR0)

The NDEDI_DTAR0 register defines a data trace address range. The address range is shared by the two ENGINEs and the CDC. All three of these clients can independently enable the range for reads and/or writes. If the start address value is greater than the end address value, all accesses are considered to have the address value outside to the data trace window 0.

NOTE

If RC0 equals 1 for a determined source, all accesses from that source will have the address value acknowledged.

Notice that the combination of the start and end addresses for data trace into one register does not follow the standard's recommended method of having separate registers for start and end addresses.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	DTSA0 [13:2]												0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	65															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	DTEA0 [13:2]												1	1
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
Index	65															

Figure 11-16. Data Trace Address Range 0 Register (NDEDI_DTAR0)

Table 11-15. NDEDI_DTAR0 Field Descriptions

Bits	Name	Description
0–1	—	Reserved.
2–13	DTSA0[0:11]	Data trace start address 0. The DTSA0 field is the start address for eTPU data trace window 0.
14–17	—	Reserved.
18–29	DTEA0[0:11]	Data trace end address 0. The DTEA0 field is the end address for eTPU data trace window 0.
30–31	—	Reserved.

11.2.1.16 Data Trace Address Range 1 Register (NDEDI_DTAR1)

The NDEDI_DTAR1 register defines a data trace address range. The address range is shared by the two ENGINEs and the CDC. All three of these clients can independently enable the range for reads and/or writes. If the start address value is greater than the end address value, all accesses are considered to have the address value outside to the data trace window 1.

NOTE

If RC1 equals 1 for a determined source, all accesses from that source will have the address value acknowledged.

Notice that the combination of the start and end addresses for data trace into one register does not follow the standard's recommended method of having separate registers for start and end addresses.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	DTSA1 [13:2]												0	0
W	▨	▨													▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	66															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	DTEA1 [13:2]												1	1
W	▨	▨													▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
Index	66															

Figure 11-17. Data Trace Address Range 1 Register (NDEDI_DTAR1)

Table 11-16. NDEDI_DTAR1 Field Descriptions

Bits	Name	Description
0–1	—	Reserved.
2–13	DTSA1[0:11]	Data trace start address 1. Start address for eTPU data trace window 1.
14–17	—	Reserved.
18–29	DTEA1[0:11]	Data trace end address 1. End address for eTPU data trace window 1.
30–31	**—**	Reserved.

11.2.1.17 Data Trace Address Range 2 Register (NDEDI_DTAR2)

The NDEDI_DTAR2 register defines a data trace address range. The address range is shared by the two ENGINEs and the CDC. All three of these clients can independently enable the range for reads and/or writes. If the start address value is greater than the end address value, all accesses are considered to have the address value outside to the data trace window 2.

NOTE

If RC2 equals 1 for a determined source, all accesses from that source will have the address value acknowledged.

Notice that the combination of the start and end addresses for data trace into one register does not follow the standard's recommended method of having separate registers for start and end addresses.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	DTSA2 [13:2]												0	0
W	▨	▨													▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	67															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	DTEA2 [13:2]												1	1
W	▨	▨													▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
Index	67															

Figure 11-18. CData Trace Address Range 2 Register (NDEDI_DTAR2)

Table 11-17. NDEDI_DTAR2 Field Descriptions

Bits	Name	Description
0–1	—	Reserved.
2–13	DTSA2[0:11]	Data trace start address 2. Start address for eTPU data trace window 2.
14–17	—	Reserved.
18–29	DTEA2[0:11]	Data trace end address 2.End address for eTPU data trace window 2.
30–31	—	Reserved.

11.2.1.18 Data Trace Address Range 3 Register (NDEDI_DTAR3)

The NDEDI_DTAR3 register defines a data trace address range. The address range is shared by the two ENGINEs and the CDC. All three of these clients can independently enable the range for reads and/or writes. If the start address value is greater than the end address value, all accesses are considered to have the address value outside to the data trace window 3.

NOTE

If RC3 equals 1 for a determined source, all accesses from that source will have the address value acknowledged.

Notice that the combination of the start and end addresses for data trace into one register does not follow the standard's recommended method of having separate registers for start and end addresses.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	DTSA3 [13:2]												0	0
W	▨	▨													▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	68															

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	DTEA3 [13:2]												1	1
W	▨	▨													▨	▨
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
Index	68															

Figure 11-19. Data Trace Address Range 3 Register (NDEDI_DTAR3)

Table 11-18. NDEDI_DTAR3 Field Descriptions

Bits	Name	Description
0–1	—	Reserved.
2–13	DTSA3[0:11]	TPU data trace start address 3. Start address for eTPU data trace window 3.
14–17	—	Reserved.
18–29	DTEA3[0:11]	eTPU data trace end address 3. End address for eTPU data trace window 3.
30–31	—	Reserved.

11.2.1.19 Unimplemented Registers

Unimplemented registers are those with client select and index value combinations other than those listed in Table 11-1. NDEDI will treat unimplemented registers like the JTAG BYPASS instruction.

11.3 Functional Description

11.3.1 NDEDI Reset Configuration

11.3.1.1 Enabling NDEDI Class 1 Operation

The NDEDI Class 1 features are always enabled after exiting the JTAG test-logic-reset state. But it is disabled while the JTAG is in this state. Thus, allowing low power mode for a production part. The registers are always reset while in the JTAG test-logic-reset state.

11.3.1.2 Enabling NDEDI Class 3 Operation

When NDEDI PCR[MCKO_EN] is asserted the NDEDI Classes 1 and 3 are enabled, otherwise the NDEDI Class 3 features will be disabled, entering in the disable-port mode, thus no trace output will be provided, and auxiliary port output pins will be disabled (driven inactive or used for an alternate function if sharing pins).[1]

11.3.2 Auxiliary Output Port

11.3.2.1 Output Message Protocol

The protocol for transmitting messages via the auxiliary port is accomplished with the $\overline{\text{MSEO}}$ functions. MDO and $\overline{\text{MSEO}}$ must be sampled by the development tool on the rising edge of MCKO.

Figure 11-20 illustrates the state diagram for $\overline{\text{MSEO}}$ transfers. All transitions not included in the figure are reserved, and are never generated by the NDEDI block.

[1]Class 1 features are still available in Disable-Port Mode.

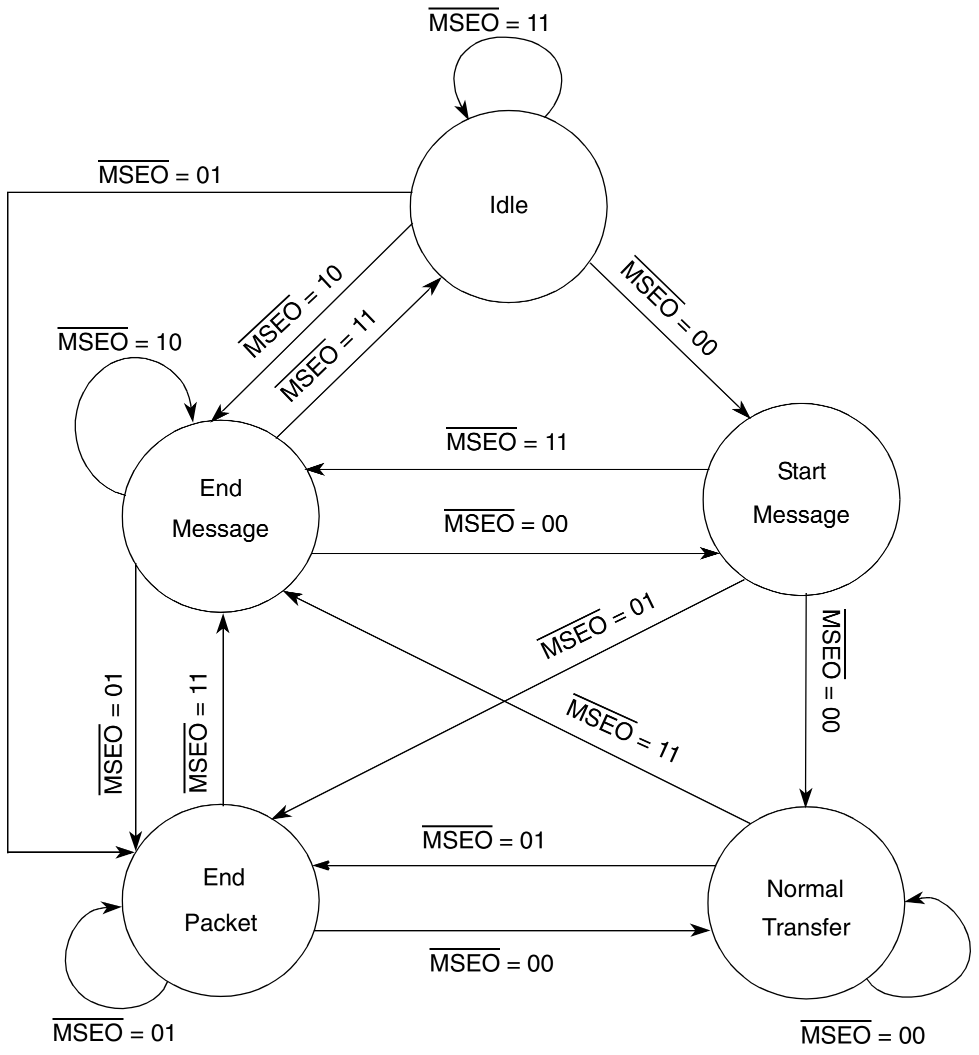


Figure 11-20. $\overline{\text{MSEO}}$ Transfers

11.3.2.2 Output Messages

Table 11-19 describes the messages that the NDEDI can transmit on the auxiliary port.

Table 11-19. NDEDI Messages

Message Name	Min. Packet Size (bits)	Max Packet Size (bits)	Packet Type	Packet Name	Packet Description
Debug Status Message	6	6	fixed	TCODE	Value = 0
	K	K	fixed	SRC	Client that is the source of the message
	32	32	fixed	STATUS	Value of the Development Status register

Table 11-19. NDEDI Messages (continued)

Message Name	Min. Packet Size (bits)	Max Packet Size (bits)	Packet Type	Packet Name	Packet Description
Ownership Trace Message	6	6	fixed	TCODE	Value = 2
	K	K	fixed	SRC	Client that is the source of the message
	16	16	fixed	PROCESS	Process ID. This value depends on the eTPU state at the beginning of a service
Data Trace, Data Write Message	6	6	fixed	TCODE	Value = 5
	K	K	fixed	SRC	Client that is the source of the message
	2	2	fixed	SIZE	Size of data access (8, 24, or 32 bits)
	1	14	variable	U-ADDR	Unique portion of the data write address. Most significant bits that have 0 values may be truncated
	1	32	variable	DATA	Data value written
Data Trace, Data Read Message	6	6	fixed	TCODE	Value = 6
	K	K	fixed	SRC	Client that is the source of the message
	2	2	fixed	SIZE	Size of data access (8, 24, or 32 bits)
	1	14	variable	U-ADDR	Unique portion of the data read address. Most significant bits that have 0 values may be truncated
	1	32	variable	DATA	Data value read
Error Message	6	6	fixed	TCODE	Value = 8
	K	K	fixed	SRC	Client that is the source of the message
	5	5	fixed	ECODE	Error Code
Data Trace, Data Write with Sync Message	6	6	fixed	TCODE	Value = 13
	K	K	fixed	SRC	Client that is the source of the message
	2	2	fixed	SIZE	Size of data access (8, 24, or 32 bits)
	1	14	variable	F-ADDR	Full address of the memory location written. Most significant bits that have 0 values may be truncated
	1	32	variable	DATA	Data value written
Data Trace, Data Read with Sync Message	6	6	fixed	TCODE	Value = 14
	K	K	fixed	SRC	Client that is the source of the message
	2	2	fixed	SIZE	Size of data access (8, 24, or 32 bits)
	1	14	variable	F-ADDR	Full address of the memory location read. Most significant bits that have 0 values may be truncated
	1	32	variable	DATA	Data value read

Table 11-19. NDEDI Messages (continued)

Message Name	Min. Packet Size (bits)	Max Packet Size (bits)	Packet Type	Packet Name	Packet Description
Watchpoint Hit Message	6	6	fixed	TCODE	Value = 15
	K	K	fixed	SRC	Client that is the source of the message
	7	7	fixed	WPHIT	Number indicating watchpoint source
Resource Full Message	6	6	fixed	TCODE	Value = 27
	K	K	fixed	SRC	Client that is the source of the message
	4	4	fixed	RCODE	Resource Code. Refer to Table 11-21
	1	H	variable	HIST	Branch/Predicate Instruction History. This packet is terminated by a stop bit set to 1 after the last history bit. This allows the tool to determine which bits are part of the history packet and which are padded zeros
Program Trace, Indirect Branch with History Message	6	6	fixed	TCODE	Value = 28
	K	K	fixed	SRC	Client that is the source of the message
	1	8	variable	I-CNT	Number of instruction units executed since the last taken branch, not taken direct branch, or predicated instruction
	1	14	variable	U-ADDR	Unique portion of the branch target address for an indirect branch. Most significant bits that have 0 values may be truncated
	1	H	variable	HIST	Branch/Predicate Instruction History. This packet is terminated by a stop bit set to 1 after the last history bit. This allows the tool to determine which bits are part of the history packet and which are padded zeros
Program Trace, Indirect Branch with History Sync Message	6	6	fixed	TCODE	Value = 29
	K	K	fixed	SRC	Client that is the source of the message
	1	8	variable	I-CNT	Number of instruction units executed since the last taken branch, not taken direct branch, or predicated instruction
	1	14	variable	F-ADDR	Full branch target address for an indirect branch. Most significant bits that have 0 values may be truncated
	1	H	variable	HIST	Branch/Predicate Instruction History. This packet is terminated by a stop bit set to 1 after the last history bit. This allows the tool to determine which bits are part of the history packet and which are padded zeros

Table 11-19. NDEDI Messages (continued)

Message Name	Min. Packet Size (bits)	Max Packet Size (bits)	Packet Type	Packet Name	Packet Description
Program Trace Correlation Message	6	6	fixed	TCODE	Value = 33
	K	K	fixed	SRC	Client that is the source of the message
	4	4	fixed	EV-CODE	Event Code
	1	8	variable	I-CNT	Number of instruction units executed since the last taken branch, not taken direct branch, or predicated instruction
	1	H	variable	HIST	Branch/Predicate Instruction History. This packet is terminated by a stop bit set to 1 after the last history bit. This allows the tool to determine which bits are part of the history packet and which are padded zeros
Program Trace, Channel Start Service Message	6	6	fixed	TCODE	Value = TCODE_CSM (Vendor-Defined Message)
	K	K	fixed	SRC	Client that is the source of the message
	5	5	fixed	S-CHAN	Number of the channel being serviced
	1	8	variable	I-CNT	Number of instruction units executed since the last taken branch, not taken direct branch, or predicated instruction
	1	14	variable	F-ADDR	Full address of the first instruction. Most significant bits that have 0 values may be truncated.
	1	H	variable	HIST	Branch/Predicate Instruction History. This packet is terminated by a stop bit set to 1 after the last history bit. This allows the tool to determine which bits are part of the history packet and which are padded zeros
Program Trace, Channel Trace Enable Message	6	6	fixed	TCODE	Value = TCODE_CTM (Vendor-Defined Message)
	K	K	fixed	SRC	Client that is the source of the message
	5	5	fixed	S-CHAN	Number of the channel being serviced
	1	14	variable	F-ADDR	Full address of the first instruction after the program trace is enabled by a watchpoint. Or, the full address of the first instruction after the Event Queue gets empty after a Queue Overrun. Most significant bits that have 0 values may be truncated
	0	5	variable	CHAN	Channel Register Value. Only sent if different from S-CHAN value

Table 11-19. NDEDI Messages (continued)

Message Name	Min. Packet Size (bits)	Max Packet Size (bits)	Packet Type	Packet Name	Packet Description
Channel Register Write Message	6	6	fixed	TCODE	Value = TCODE_CWM (Vendor-Defined Message)
	K	K	fixed	SRC	Client that is the source of the message
	1	5	variable	CHAN	Value written to the channel register

Table 11-20 describes the error code encodings for the error messages generated by the NDEDI block.

Table 11-20. Error Codes Encodings (ECODE)

Error Code	Description
0b00000	Ownership trace overrun
0b00001	Program trace overrun
0b00010	Data trace overrun
0b00110	Watchpoint overrun
0b00111	Program and/or Data and/or Ownership Trace Overrun
0b01000	Program trace and/or data trace and/or ownership trace and/or watchpoint overrun
0b11000	Debug status overrun
0b11001	Debug status overrun and/or program and/or data and/or ownership trace overrun
0b11010	Debug status and/or program trace and/or data trace and/or ownership trace and/or watchpoint overrun

Table 11-21 describes the resource code encodings for the resource full messages.

Table 11-21. Resource Codes Encodings

Resource Code	Resource	DATA Packet Count	DATA Packet Value
0b0000	Program Trace, Sequential Counter	1	Branch/predicate instruction history. This packet is terminated by a stop bit set to 1 after the last history bit. The value that causes the counter to fill is always the same and can be determined without sending the value
0b0001	Program Trace, Branch/Predicate History	1	Branch/predicate instruction history. This packet is terminated by a stop bit set to 1 after the last history bit

Message formatting is performed by the message formatter block. Raw messages read from the message queues are independent of the number of MDO pins implemented.

Table 11-21 shows the various message formats and the location of variable length packets. Notice that for variable-length packets, the transmitted size of the packet is determined from the range of the least significant bit to the most significant non-zero-valued bit (i.e. most significant 0 value bits are not transmitted).

Message	TCODE	Packet #1	Packet #2	Packet #3	Packet #4	Packet #5	Packet #6	Min. Size[1] (bits)	Max Size[2] (bits)
Debug Status Message	0	Fixed = K	Fixed =32	NA	NA	NA	NA	38+K	38+K
Ownership Trace Message	2	Fixed = K	Fixed =16	NA	NA	NA	NA	22+K	22+K
Data Trace, Data Write Message	5	Fixed = K	Fixed[3] = 2	Variable Min = 1 Max = 14	Variable Min = 1 Max = 32	NA	NA	10+K	54+K
Data Trace, Data Read Message	6	Fixed = K	Fixed[3] = 2	Variable Min = 1 Max = 14	Variable Min = 1 Max = 32	NA	NA	10+K	54+K
Error Message	8	Fixed = K	Fixed = 5	NA	NA	NA	NA	11+K	11+K
Data Trace, Data Write with Sync Message	13	Fixed = K	Fixed[3] = 2	Variable Min = 1 Max = 14	Variable Min = 1 Max = 32	NA	NA	10+K	54+K
Data Trace, Data Read with Sync Message	14	Fixed = K	Fixed[3] = 2	Variable Min = 1 Max = 14	Variable Min = 1 Max = 32	NA	NA	10+K	54+K
Watchpoint Hit Message	15	Fixed = K	Fixed = 7	NA	NA	NA	NA	13+K	13+K
Resource Full Message	27	Fixed = K	Fixed = 4	Variable Min = 1 Max = H	NA	NA	NA	11+K	10+H+K
Program Trace, Indirect Branch w/ History Message	28	Fixed = K	Variable Min = 1 Max = 8	Variable Min = 1 Max = 14	Variable Min = 1 Max = H	NA	NA	9+K	28+H+K
Program Trace, Indirect Branch w/ History Sync Message	29	Fixed = K	Variable Min = 1 Max = 8	Variable Min = 1 Max = 14	Variable Min = 1 Max = H	N/A	NA	9+K	28+H+K
Program Trace, Correlation Message	33	Fixed = K	Fixed=4	Variable Min = 1 Max = 8	Variable Min = 1 Max = H	NA	NA	12+K	18+H+K
Program Trace, Channel Start Service Message	TCODE _CSM	Fixed = K	Fixed = 5	Variable Min = 1 Max = 8	Variable Min = 1 Max = 14	Variable Min = 1 Max = H	NA	14+K	33+H+K
Program Trace, Channel Trace Enable Message	TCODE _CTM	Fixed = K	Fixed=5	Variable Min = 1 Max = 14	Variable Min = 0 Max = 5	NA	NA	12+K	30+K
Channel Register Write Message	TCODE _CWM	Fixed = K	Variable Min = 1 Max = 5	NA	NA	NA	NA	7+K	11+K

NOTES:
1. Minimum information size. The actual number of bits transmitted depends on the number of MDO pins.

Figure 11-21. Message Packet Sizes

Message	TCODE	Packet #1	Packet #2	Packet #3	Packet #4	Packet #5	Packet #6	Min. Size[1] (bits)	Max Size[2] (bits)

2. Maximum information size. The actual number of bits transmitted depends on the number of MDO pins.

3. Packet is fixed and NDEDI always transmits 2 bits, but other sources sharing the auxiliary port may have a different packet size. Packet size can always be determined by the value of the SRC packet.

Figure 11-21. Message Packet Sizes

The double edges in Figure 11-21 indicate the starts and ends of messages. The shaded edges in Figure 11-21 indicate the end of a variable length packet which is not also the end of the message. Packets without shaded areas between them are grouped into super packets and are transmitted together without end-of-packet indications between them.

11.3.2.3 Rules of Messaging

- A variable-length packet within a message must end on a port boundary (Port boundaries depend on the number of MDO pins active with the current reset configuration)
- A variable-length packet may start within a port boundary only when following a fixed-length packet.
- Superblocks must end on a port boundary
- When a variable-length packet is sized such that it does not end on a port boundary, it is necessary to extend and zero fill the remaining bits after the highest order bit so that it can end on a port boundary
- Multiple fixed-length packets may start and/or end on the same clock
- When any packet follows a variable-length packet, it must start on a port boundary
- The packet containing the TCODE number is always transferred out first, followed by subsequent packets of information
- Within a packet, the least significant bits are shifted out first. Figure 11-22 shows the transmission sequence of a message that is made up of a TCODE followed by two packets

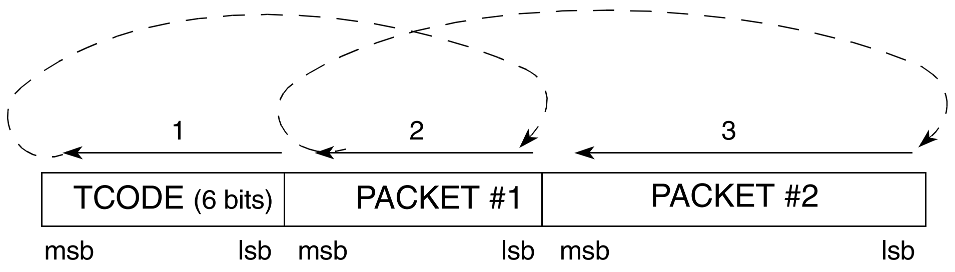


Figure 11-22. Transmission Sequence of Messages

11.3.2.4 Examples

The following are examples of branch trace and data trace messages.

Figure 11-23 illustrates how the indirect branch with history public message is transmitted. Notice that the example uses a 4-bit MDO port. For this example, 4 bits are shifted out in the I-CNT packet, 5 bits are shifted out in the HIST packet, and 11 bits are shifted out for the UADDR packet. Notice also that T0, S0, H0, I0, and A0 are the least significant bits where:

Tx= TCODE packet bits (Fixed)
Sx= SRC packet bits (Fixed)
Hx = HIST packet bits (Variable)
Ix = I-CNT packet bits (Variable)
Ax = UADDR packet bits (Variable)

Clock	MDO[3:0]				$\overline{\text{MSEO}}$[1:0]	State
	3	2	1	0		
0	X	X	X	X	X	End Message or Idle
1	T3	T2	T1	T0	00	Start Message
2	S1	S0	T5	T4	00	Normal Transfer
3	I2	I1	I0	S2	00	Normal Transfer
4	0	0	0	I3	01	End Packet
5	A3	A2	A1	A0	00	Normal Transfer
6	A7	A6	A5	A4	00	Normal Transfer
7	0	A10	A9	A8	01	End Packet
8	H3	H2	H1	H0	00	Normal Transfer
9	0	0	0	H4=1	11	End Message

Figure 11-23. Indirect Branch with History Message (4 MDO pins)

Figure 11-24 shows the same message being sent using 8 MDO pins.

Clock	MDO[7:0]								$\overline{\text{MSEO}}$[1:0]	State
	7	6	5	4	3	2	1	0		
0	X	X	X	X	X	X	X	X	X	End Message or Idle
1	S1	S0	T5	T4	T3	T2	T1	T0	00	Start Message
2	0	0	0	I3	I2	I1	I0	S2	01	End Packet
4	A7	A6	A5	A4	A3	A2	A1	A0	00	Normal Transfer
5	0	0	0	0	0	A10	A9	A8	01	End Packet
3	0	0	0	H4=1	H3	H2	H1	H0	11	End Message

Figure 11-24. Indirect Branch with History Message (8 pins)

Figure 11-25 illustrates how a data write with sync message is transmitted. *Notice that the example uses a 16-bit MDO port***.** For this example, 7 bits are sent for the F-ADDR packet, and 5 bits are sent for the DATA packet. Notice also that T0, S0, A0, and D0 are the least significant bits where:

Tx = TCODE packet bits (Fixed)
Sx = SRC packet bits (Fixed)
Zx = SIZE packet bits (Fixed)
Dx = DATA packet bits (Variable)

Ax = F-ADDR packet bits (Variable)

Clock	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	$\overline{\text{MSEO}}$[1:0]	State
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	End Message or Idle
1	A4	A3	A2	A1	A0	Z1	Z0	S2	S1	S0	T5	T4	T3	T2	T1	T0	00	Start Message
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A6	A5	01	End Packet
3	0	0	0	0	0	0	0	0	0	0	0	D4	D3	D2	D1	D0	11	End Message

MDO[15:0]

Figure 11-25. Data Write with Sync Message (16 pins)

11.3.2.5 Temporal Ordering of Transmitted Messages

All Messages are sent out in the sequence their related event actually occurred. The eTPU interface with the NDEDI provides the eTPU signals to be monitored and captured. At the occurrence of any event that would cause a message, a snapshot of all information needed by the Message Formatter to generate a message is queued. Doing so, several events from either Engines or CDC can be queued at the same time.

Thus, as soon as the auxiliary port is granted, the event queue is read and the appropriate messages are formatted and sent. If more than one messages are to be sent, the message formatter respects the following priority: messages related to error, debug status, watchpoints, program trace and data trace.

NDEDI maintain temporal ordering of messages relative to ENGINE1, ENGINE2 and CDC, but not relative to messages generated by other clients sharing the auxiliary port.The time slot transition (TST) is considered to be an active state, and since data are accessed at the SPRAM, it is possible that data trace events are recognized within TST.[1] For further information about TST, refer to the .

11.3.3 Microcode Development Support

The NDEDI contains a number of hardware hooks that aid in the development of microcode for the eTPUs. This features described in this section make the eTPUs compliant with Nexus Class 1 features.

The main features provided are the abilities to:

- Read and write eTPU internal registers in debug mode
- Read and write SPRAM in debug mode
- Enter a debug mode at reset negation
- Enter a debug mode during normal execution
- Single step instructions and re-enter debug mode

[1]Due to eTPU operation, speculative accesses may also happen within TST and will be traced.

- Stop program execution on instruction/data breakpoints or channel service breakpoints and enter debug mode
- Set breakpoints or watchpoints
- Execute external microcode instruction in debug mode
- Read/write the microprogram counter register value in debug mode

Refer to Section 11.4, “Initialization/Application Information,” on how the following functions provide the features listed above.

11.3.4 Debug Status

The NDEDI module provides the debug status via the auxiliary port, as defined by the IEEE-ISTO 5001-2002 standard.

11.3.4.1 Messaging

The NDEDI block provides debug status messaging using IEEE-ISTO 5001-2002 standard-defined public messages. When the development status register changes, a debug status change event is sent to the event queue. If the debug status change condition occurs while the event queue is not enabled for storing snapshots, a debug status overrun message is generated.

The debug status message has the format shown in Figure 11-26.

[6 bits]	[K bits]	[32 bits]
TCODE (0)	SRC	DS

Length = 38+K bits

Figure 11-26. Debug Status Message Format

The DS packet correspond to the ENGINEx DS register value described in Figure 11-4.

11.3.4.2 Error Messages

A debug status overrun error event is queued if a debug status event occurs while the event queue is not enabled for storing snapshots. The error event is stored as soon as the event queue is enabled which happens when the queue becomes empty.

The error message has the format shown in Figure 11-27.

[6 bits]	[K bits]	[5 bits]
TCODE (8)	SRC	ECODE (0b11000)

Length = 11+K bits

Figure 11-27. Debug Status Error Message Format

11.3.4.3 Synchronization

Upon the exit of debug mode, the next program and data trace messages are synchronization messages. The exit of debug mode by one of the engines causes data and program synchronization for that engine only.

11.3.4.4 Timing Diagrams

Note that all of the following timing diagrams assume a 3-bit SRC size and a 16-bit MDO port. The state variable is not a signal, but instead is derived from $\overline{\text{MSEO}}$. It is included for clarity. Refer to Figure 11-20 for $\overline{\text{MSEO}}$ state diagram.

The following abbreviations are used for the state variable in the diagrams:

- ID = Idle
- SM = Start Message
- NT = Normal Transfer
- EP = End Packet
- EM = End Message

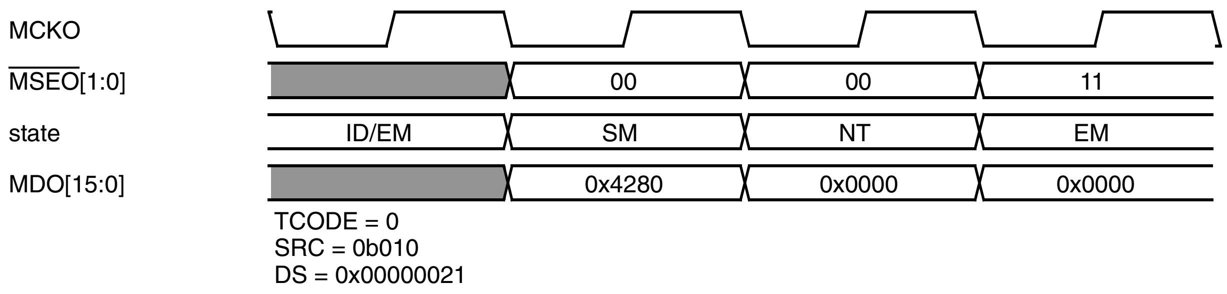


Figure 11-28. Debug Status Message

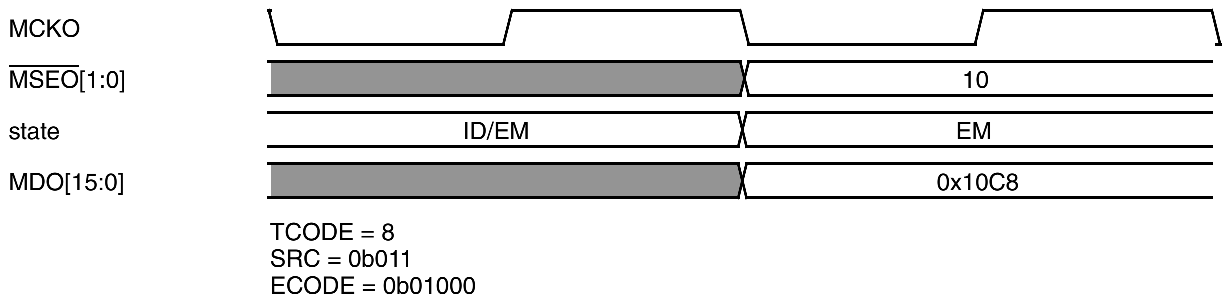


Figure 11-29. Debug Status Overrun Error Message

11.3.5 Ownership Trace

Ownership trace provides a macroscopic view of the eTPU program flow. This is done by generating an ownership trace event at the start of each channel service.

Each engine within the eTPU system generates ownership traces independently, and the events are queued in the order in which they occur.

11.3.5.1 Messaging

Ownership trace information is transmitted via the auxiliary port using an ownership trace message (OTM).

Ownership trace information is transmitted in the format shown in Figure 11-30.

[6 bits]	[K bits]					[16 bits]					
TCODE (2)	SRC	Channel (5 bits)	HSR (3 bits)	Link (1 bit)	M1 (1 bit)	M2 (1 bit)	T1 (1 bit)	T2 (1 bit)	Pin (1 bit)	CF1 (1 bit)	CF0 (1 bit)

Length = 22+K bits

Figure 11-30. Ownership Trace Message Format

Notice that the channel number and entry points number in Figure 11-30 are concatenated to form the variable length PROCESS packet defined by the IEEE-ISTO 5001-2002 standard.

11.3.5.2 OTM Flow

A channel generates an ownership trace message when service starts if ownership trace is enabled for the eTPU via the development control register. If ownership trace is enabled and an engine becomes idle, NDEDI will transmit a special OTM with the PROCESS packet set to 0.

11.3.5.3 Timing Diagram

Figure 11-31 shows an example of an ownership message being transmitted on the auxiliary port. The diagram assumes a 3-bit SRC size and a 16-bit MDO port. The state variable is not a signal but instead, it is derived from $\overline{\text{MSEO}}$. It is included for clarity. Refer to Figure 11-20 for $\overline{\text{MSEO}}$ state diagram.

The following abbreviations are used for the state variable in the diagrams:

- ID = Idle
- SM = Start Message
- NT = Normal Transfer
- EP = End Packet
- EM = End Message

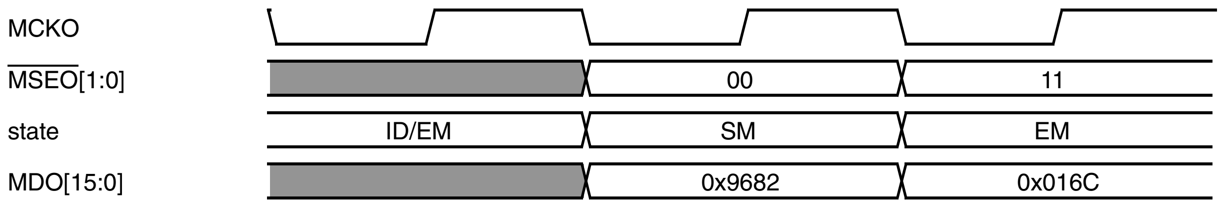

MCKO
$\overline{\text{MSEO}}$[1:0]
state
MDO[15:0]
00
11
ID/EM
SM
EM
0x9682
0x016C
TCODE = 2
SRC = 0b010
PROCESS = {0b10110, 0b110, 0b0, 0b1, 0b0, ob0, 0b1, 0b0, 0b1, 0b1}

Figure 11-31. Ownership Trace Message

11.3.6 Program Trace

This section details the program trace mechanism supported by the NDEDI for the dual Engine system. Program trace is implemented via a combination of Branch Trace Messaging (BTM) and Ownership Trace Messaging (OTM) as per the IEEE-ISTO 5001-2002 standard definition.

11.3.6.1 Branch Trace Messaging

Branch trace messaging facilitates program trace by providing the following types of information:

- Messages for the start of channel services, each containing the full address of the first instruction to execute and history and sequential count information left over from the previously traced channel service. This message is always synchronizing
- Messages to indicate that the program trace was enabled in the middle of a thread . It includes the address of the enable point, the value of the serviced channel and the CHAN reg value only if it differs from the serviced channel value
- Messages to indicate that the internal history buffer has filled and been reset. The current value of the history buffer is included in each of these messages
- Messages to indicate that the internal sequential instruction counter has overflow and been reset. The current value of the history buffer is included in each of these messages
- Messages for taken indirect branches, each including a branch/predicate instruction[1] history packet, sequential instruction count, and the unique portion of the branch target address. Indirect branch messages that are generated with pending synchronization event include the full branch target address (instead of the unique portion of the address)
- Messages to indicate the entry into debug mode, the entry into low power mode, or the end of channel service with no new service pending

[1]Predicate Instructions depend on the AS/CE field within eTPU instructions.

- Messages for writes to the CHAN register, indicating the value assigned to the register

The address packet of a program trace address excludes the 2 least significant bits of the byte address because they are zero for all instruction addresses.

11.3.6.2 Branch Trace Message Formats

There are seven types of messages used by the eTPU for branch tracing. These types are shown in Table 11-22.

Table 11-22. eTPU BTM Messages

Message	Synchronizing	Public/Private
Resource full message	No	Public
Indirect branch with history message	No	Public
Indirect branch with history synchronization message	Yes	Public
Program trace correlation message	No	Public
Channel start service message	Yes	Private
Channel trace enable message	Yes	Private
Channel register write message	No	Private

The occurrence of any of the following conditions requires subsequent synchronization:

- Exit of system reset
- Exit of low power mode
- Exit of debug mode
- Forced End executed in eTPU engine
- Write to the development control register or the program trace channel enable register[1]
- 255 branch trace messages of the same are queued without synchronization
- Assertion of the event in ($\overline{\text{EVTI}}$) pin, depending on how the development control register is configured
- A watchpoint occurrence[2]
- A queue overrun

The message that follows any of these synchronizing events will be a synchronizing message if it is not a resource full message or a channel register write message which have no synchronizing information. If no code is executing, the next message is always a channel start service message that is synchronizing.

[1]Only the source that have the register written will generate a synchronization message.

[2]A Watchpoint occurrence for one engine generates synchronization message for that engine only.

11.3.6.2.1 Resource Full Messages

The resource full message is generated when the branch/predicate history buffer is full or the sequential instruction counter overflows.

For messages caused by the branch/predicate history buffer filling, the current value of that buffer is transmitted as part of the resource full message. This information can be concatenated by the development tool with the branch/predicate history information from subsequent messages to obtain the complete branch history for a message. The HIST value is reset by this message, and the I-CNT value is reset as a result of a bit being added to the history buffer.

For messages caused by sequential instruction counter overflow, the HIST packet is also transmitted. This occurrence indicates that exactly 256 sequential instructions have been executed without a history bit being recorded or a message (other than CHAN_WRITE) being generated. The HIST and I-CNT values are reset by this message.

The resource full message has the format shown in Figure 11-32.

[6 bits]	[K bits]	[4 bits]	[1-H bits]
TCODE (27)	SRC	RCODE	HIST

Min Length = 11+K bits
Max Length = 10+H+K bits

Figure 11-32. Resource Full Message Format

11.3.6.2.2 Indirect Branch with History Messages

The indirect branch with history message is generated on a taken indirect branch with no synchronization event pending. The program trace indirect branch with history message has the format shown in Figure 11-33.

[6 bits]	[K bits]	[1-8 bits]	[1-13 bits]	[1-H bits]
TCODE (28)	SRC	I-CNT	U-ADDR	HIST

Min Length = 9+K bits
Max Length = 27+H+K bits

Figure 11-33. Indirect Branch with History Message Format

11.3.6.2.3 Indirect Branch with History Synchronization Messages

The indirect branch with history synchronization message is generated on a taken indirect branch with a synchronization event pending.

The program trace indirect branch with history synchronization message has the format shown in Figure 11-34.

[6 bits]	[K bits]	[1-8 bits]	[1-13 bits]	[1-H bits]
TCODE (29)	SRC	I-CNT	F-ADDR	HIST

Min Length = 9+K bits
Max Length = 27+H+K bits

Figure 11-34. Indirect Branch with History Synchronization Message Format

11.3.6.2.4 Program Trace Correlation Message

The program trace correlation message is generated at the end of an eTPU channel service if there is no new service, if there is a new service which channel is not enabled for tracing, if program trace is disabled in the middle of a thread being traced, upon entry into debug mode, or upon entry into low power mode.

The message contains the history and sequential instruction count information generated during the previous traced service since the last message generation.

The correlation message also contains an event code that indicates the event responsible for the generation of the message. Event code encodings are shown in Table 11-23.

Table 11-23. Event Code Encodings

EV-CODE	Description
0b0000	Entry into Debug Mode
0b0001	Entry into Low Power Mode
0b0010	Program Trace disabled in the middle of a thread
0b1110	End Channel Service (no new service)
All Others	Reserved

The program trace correlation message has the format shown in Figure 11-35.

[6 bits]	[K bits]	[4 bits]	[1-8 bits]	[1-H bits]
TCODE (33)	SRC	EV-CODE	I-CNT	HIST

Min Length = 12+K bits
Max Length = 18+H+K bits

Figure 11-35. Program Trace Correlation Message Format

11.3.6.2.5 Channel Start Service Message

The channel start service message is generated at the start of an eTPU channel service. The message contains the history and sequential instruction count information generated during the previous traced service since the last message generation. The message also contains the address of the first instruction of the current service.

The program trace channel start service message has the format shown in Figure 11-36.

[6 bits]	[K bits]	[5 bits]	[1-8 bits]	[1-13 bits]	[1-H bits]
TCODE (TCODE_CSM)	SRC	S-CHAN	I-CNT	F-ADDR	HIST

Min Length = 14+K bits
Max Length = 32+K bits

Figure 11-36. Channel Start Service Synchronization Message Format

11.3.6.2.6 Channel Trace Enable Message

If the program trace is enabled in the middle of a thread, the channel trace enable message is sent indicating to the development tool the number of the channel being serviced and the address of the current instruction.[1] If the CHAN register value differs from the serviced channel, the CHAN field is sent at the end of the message indicating the current CHAN register value.[2]

The program trace channel trace enable message has the format shown in Figure 11-37.

[6 bits]	[K bits]	[5 bits]	[1-13 bits]	[0-5 bits]
TCODE (TCODE_CTM)	SRC	S-CHAN	F-ADDR	CHAN

Min Length = 14+K bits
Max Length = 32+K bits

Figure 11-37. Channel Trace Enable Synchronization Message Format

11.3.6.2.7 Channel Register Write Messages

The channel register write message is generated when the CHAN register is written to by the microcode. This message is not necessary to reconstruct program flow and generation is enabled/disabled by the CHW bit in the ENGINEx development control register. Program trace must be enabled for the channel that started the current service for this message to be generated.

The channel register write message has the format shown in Figure 11-38

[6 bits]	[K bits]	[1-5 bits]
TCODE (TCODE_CWM)	SRC	CHAN

Min Length = 7+K bits
Max Length = 11+K bits

Figure 11-38. Channel Register Write Message Format

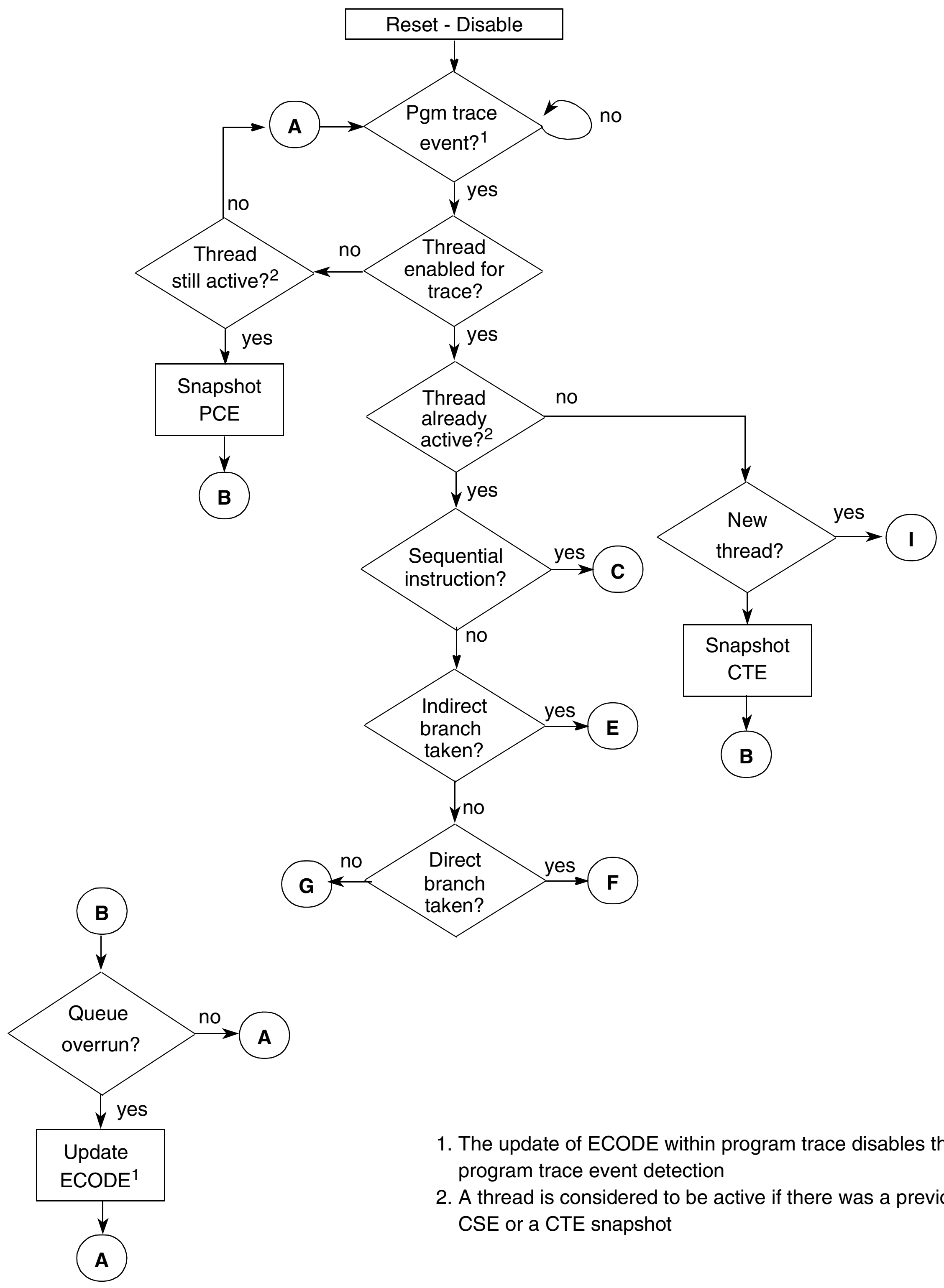
11.3.6.3 Branch Trace Messaging Operation

Figure 11-39 and Figure 11-40 show the basic flow used to generate branch trace messages for the eTPUs. The following acronyms are used in the figures:

[1]The message does not contains the history and sequential instruction count information since the trace of this channel was disabled until there.

[2]Usually the CHAN register equals the serviced channel.

- CHAN—CHAN register in eTPU execution unit
- S-CHAN—Channel being serviced
- HIST—Nexus shift register whose value is transmitted in the HIST packet of many BTMs
- I-CNT—Nexus sequential instruction counter whose value is transmitted in the HIST packet of many BTMs
- CWE— Channel write event
- IHE—Indirect branch with history event
- IHS—Indirect branch with history synchronization event
- CSE—Channel start service event
- CTE—Channel trace enable event
- RFE—Resource full event
- PCE—Program trace correlation event



1. The update of ECODE within program trace disables the program trace event detection
2. A thread is considered to be active if there was a previous CSE or a CTE snapshot

Figure 11-39. Branch Trace Message Generation (Part 1)

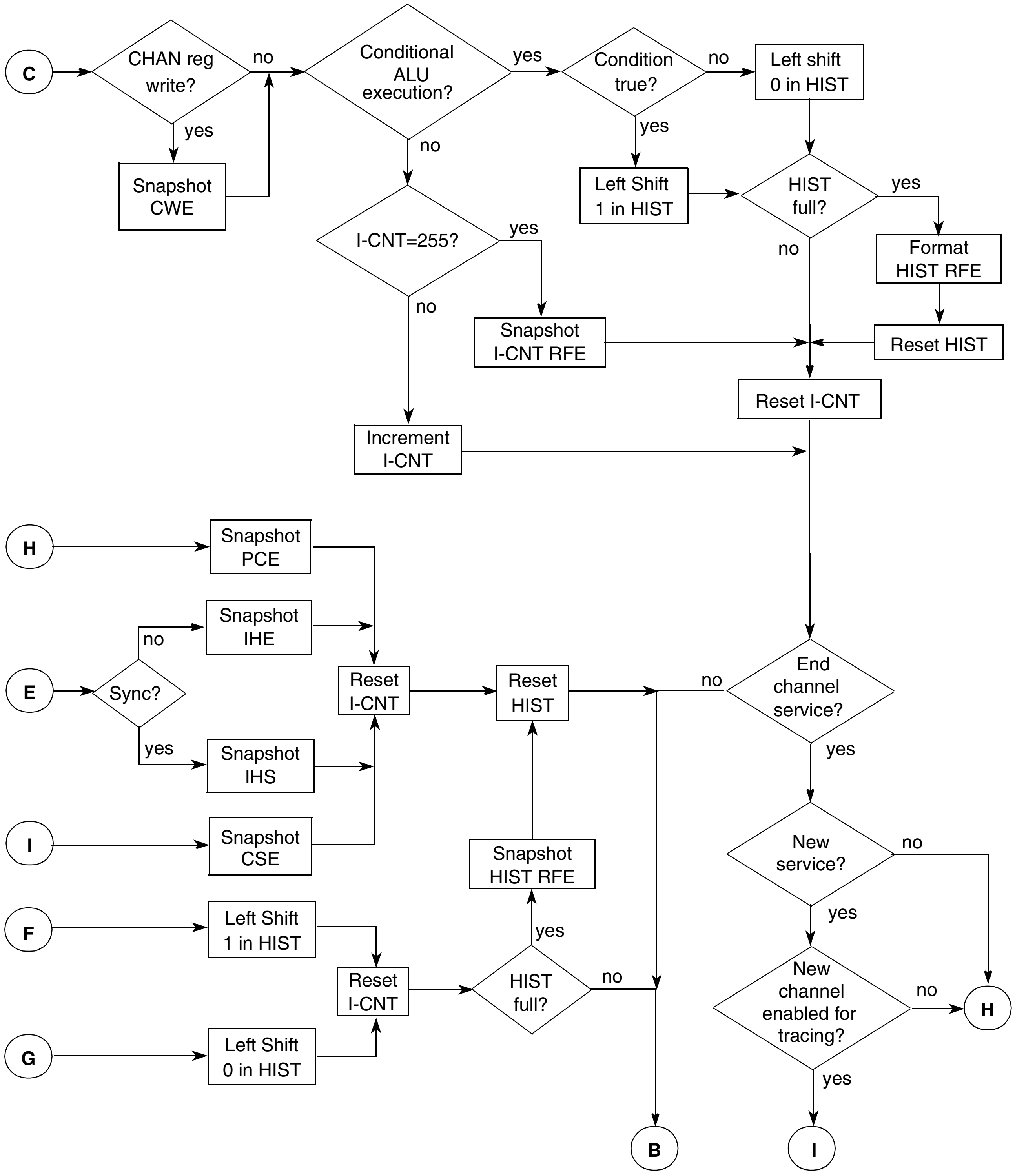


Figure 11-40. Branch Trace Message Generation (Part 2)

11.3.6.3.1 Relative Addressing

The relative address feature is compliant with the IEEE-ISTO 5001-2002 standard recommendations and is designed to reduce the number of bits transmitted for addresses of program trace messages.

The address transmitted for a determined source is relative to the address of the previous branch trace message sent for that source. It is generated by XOR'ing the new address with the previous address, and then using only the results up to the most significant '1' in the result. To recreate this address, an XOR of the (most-significant 0-padded) message address with the previously decoded address gives the current address. Figure 11-41 shows how a relative address is generated and how it can be used to recreate the original address.

```
Engine 1: Current Address (A1) = 0x65DC,
Engine 2: Current Address (B1) = 0x7238

Next Message Relative to Engine 1: Address (A2) = 0x1AF8

    Address Generation:

    A1 = 110 0101 1101 11 00
    A2 = 001 1010 1111 10 00
 A1 ^ A2 = 111 1111 0010 01 00

    Address Message (M1) = 111 1111 0010 01


    Address Recreation:

    A1 ^ M1 = A2

    A1   = 110 0101 1101 11 00
    ME1 = 111 1111 0010 01
A1 ^ ME1 = 001 1010 1111 10 00 = A2

    Next Message Relative to Engine 2: Address (B2) = 0x54C8

    Address Generation for Engine 2:

    B1 = 111 0010 0011 10 00
    B2 = 101 0100 1100 10 00
 B1 ^ B2 = 010 0110 1111 00 00

    Address Message (ME2) = 10 0110 1111 00

    Address Recreation:

    B1 ^ M1 = B2

    B1   = 111 0010 0011 10 00
    ME2 = 010 0110 1111 00
B1 ^ ME2 = 101 0100 1100 10 00 = B2
```

Figure 11-41. Relative Address Generation and Recreation

11.3.6.3.2 Enabling Program Trace

Program trace for the eTPU is enabled on a channel-by-channel basis. This selective enabling allows the tool more control over how much information gets transmitted.

For program trace information to be generated for a channel, the appropriate bit in the program trace channel enable register must be set and program trace must be enabled via the TM field in the DC register or via a watchpoint trigger.

Any combination of channels can be enabled for trace at the same time by configuring the program trace channel enable register. See Section 11.2.1.9, “ENGINEn Program Trace Channel Enable Register (NDEDI_ENGINEn_PTCE)” for more details.

Note that since the registers may be changed in the middle of a thread, program trace may be enabled or disabled in the middle of a thread.

If program trace for a determined channel is enabled in the middle of a service routine, a Channel Trace Enable Message is sent to the development tool indicating the number of the channel being serviced and the address of the current instruction. Otherwise, normal program trace begins only on the following service for that channel.If program trace for a determined channel is disabled in the middle of a service routine, a Program Correlation Message is sent to the development tool indicating that the trace is disabled and informing the HIST and I-CNT values.

11.3.6.3.3 Branch/Predicate Instruction History

The HIST packet provides a history of direct branch and predicated instruction execution used for reconstructing the program flow. This packet is implemented as a left-shifting shift register. The register is always pre-loaded with a value of 1. This bit acts as a stop bit so the tool can determine which bit is the end of the history information. The pre-loaded 1 bit itself is not part of the history information but is transmitted with the packet.

A value of 1 is shifted into the HIST packet on a taken direct branch (conditional or unconditional) and on a predicate instruction whose predicate condition evaluated as true. A value of 0 is shifted into the HIST packet on a not taken direct branch and on a predicate instruction whose predicate condition evaluated as false.

A direct branch is a branch whose target address is hard-coded into the microcode instruction at compile time. An indirect branch is a branch whose target address depends on run time conditions. The indirect branches in the eTPU are return from subroutine, dispatch jump, and dispatch call.

For the eTPU, predicated instructions are those that execute ALU operations conditionally based on the value of the C, Z, or N flags. This conditional execution is controlled by the value of the AS/CE packet in the eTPU microcode instruction formats A3, B4, B5, and B6.

11.3.6.3.4 Sequential Instruction Count

The I-CNT packet, present in many of the program trace messages, represents the number of eTPU microcode instructions executed since the start of channel service, last taken direct branch, last not-taken direct branch, last taken indirect branch, or last predicated instruction.[1] In other words, I-CNT is reset whenever it is transmitted within a message or a bit is recorded to HIST.

When a sequential instruction is executed and the current I-CNT value is 255, a resource full message is transmitted to tell the tool that the I-CNT has reached a value of 256. The current HIST packet is transmitted as part of this message and reset inside the NDEDI. The HIST and I-CNT information in the resource full message are combined with information from subsequent messages to provide the development tool a full picture of the program flow.

If additional HIST bits are recorded between a resource full message for the I-CNT and the next program trace message, the I-CNT overflow can be ignored by the tool. In this case, the HIST packet transmitted by any resource full messages and the next program trace are concatenated to provide the full HIST information.

Multiple I-CNT resource full messages can be received between other program trace messages. In these cases, the HIST packets are concatenated, and the I-CNT values are added only when no HIST bits were recorded between the resource full messages.

11.3.6.3.5 Interleaved ENGINE1 and ENGINE2 messages

Both ENGINE1 and ENGINE2 may be enabled for program trace at the same time. Each Engine microcode instruction takes two system clocks to execute. The dual Engine system alternates instruction completion between ENGINE1 and ENGINE2 so that one module never completes an instruction in the same clock as the other.The NDEDI block formats and queues ENGINE1 and ENGINE2 messages in the order they are generated. There is no way for the development tool to infer any other temporal information from the eTPU messages.

11.3.6.4 Timing Diagrams

Note that all of the following timing diagrams assume a 3-bit SRC size and either a 4-bit MDO port or a 16-bit MDO port. The state variable is not a signal, but instead is derived from $\overline{\text{MSEO}}$. It is included for clarity. Refer to Figure 11-20 for $\overline{\text{MSEO}}$ state diagram.

The following abbreviations are used for the state variable in the diagrams:

- ID = Idle
- SM = Start message
- NT = Normal transfer
- EP = End packet
- EM = End message

[1]The eTPU does not have a conditional indirect branch. Thus, there is no not-taken indirect branch.

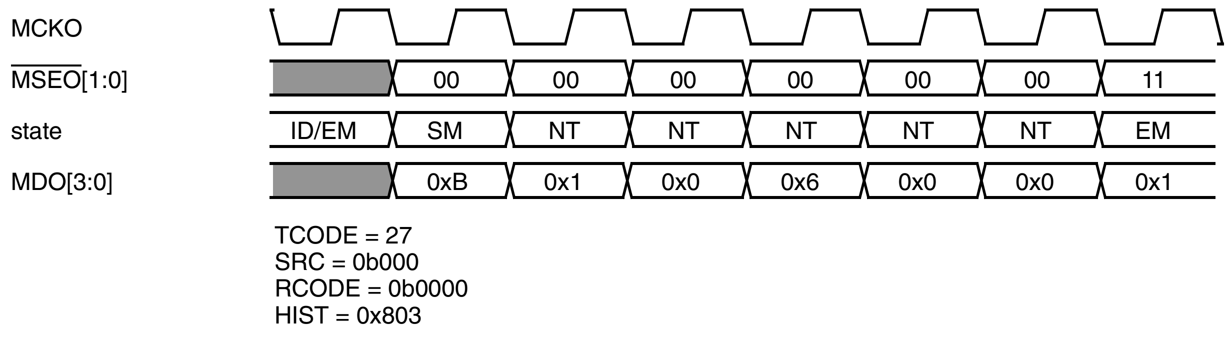


Figure 11-42. Resource Full Message, I-CNT Overflow

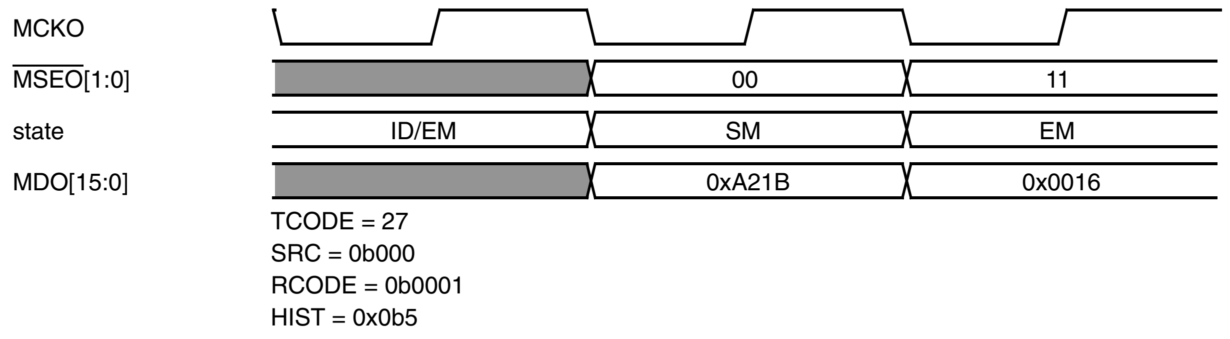


Figure 11-43. Resource Full Message, HIST Buffer Full

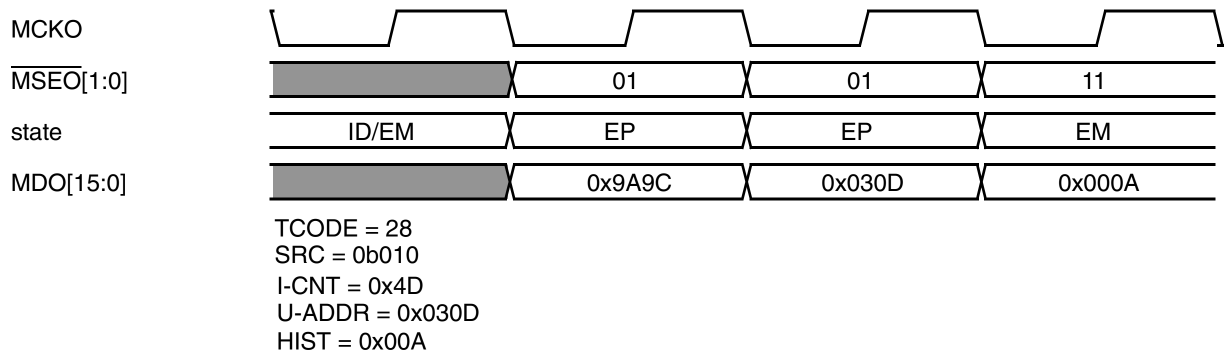


Figure 11-44. Indirect Branch with History Message

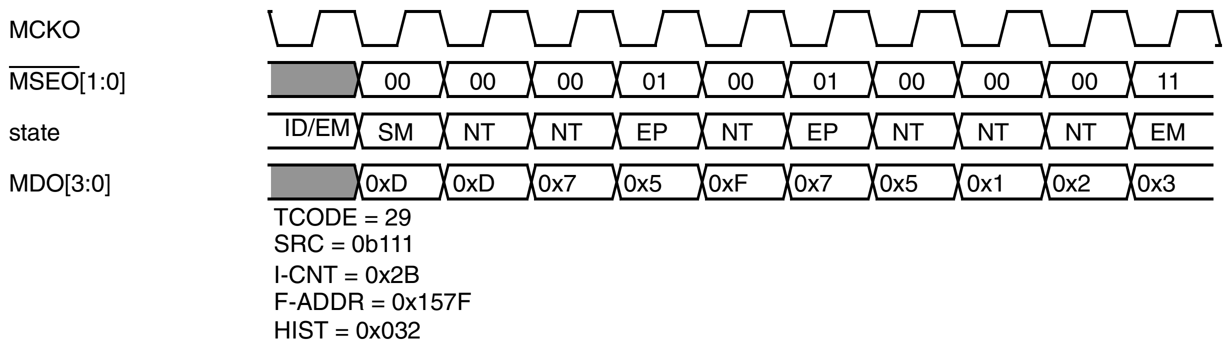


Figure 11-45. Indirect Branch with History Synchronization Message

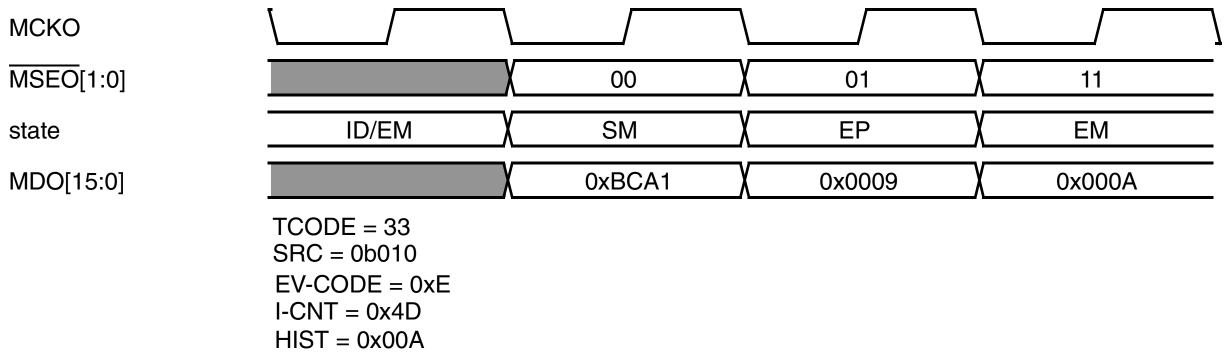


Figure 11-46. Program Trace Correlation Message

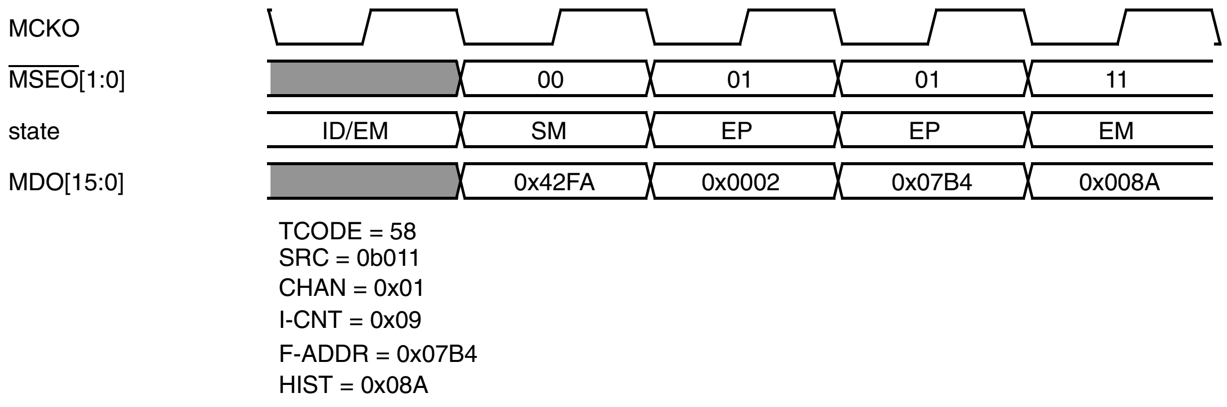


Figure 11-47. Channel Start Service Message

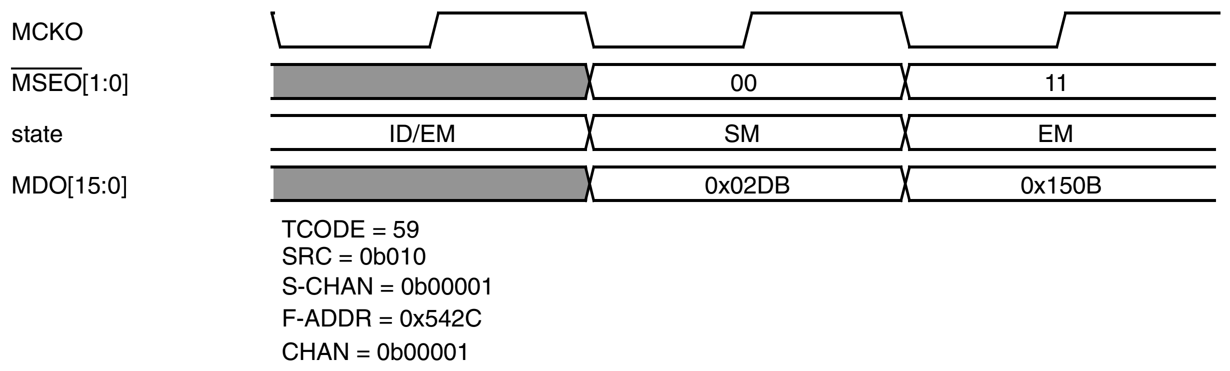


Figure 11-48. Channel Trace Enable Message with the same value for CHAN and S-CHAN

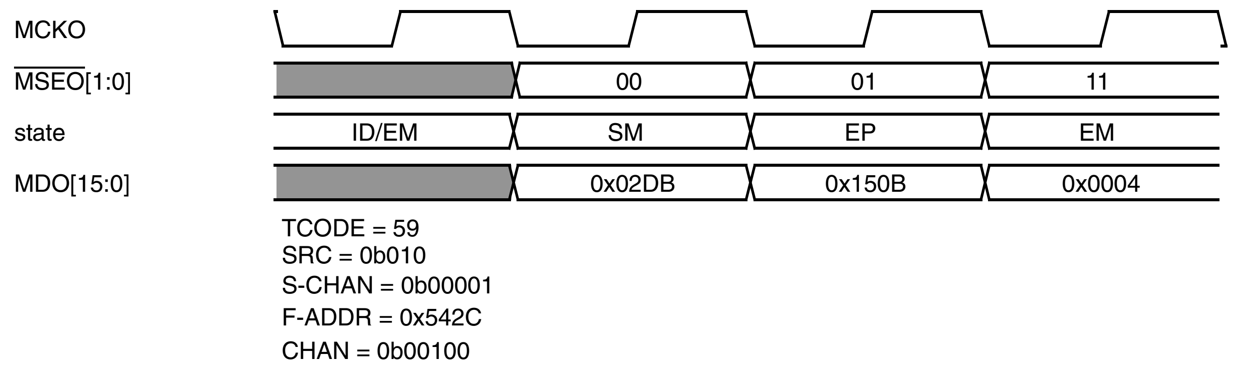


Figure 11-49. Channel Trace Enable Message with different values for CHAN and S-CHAN

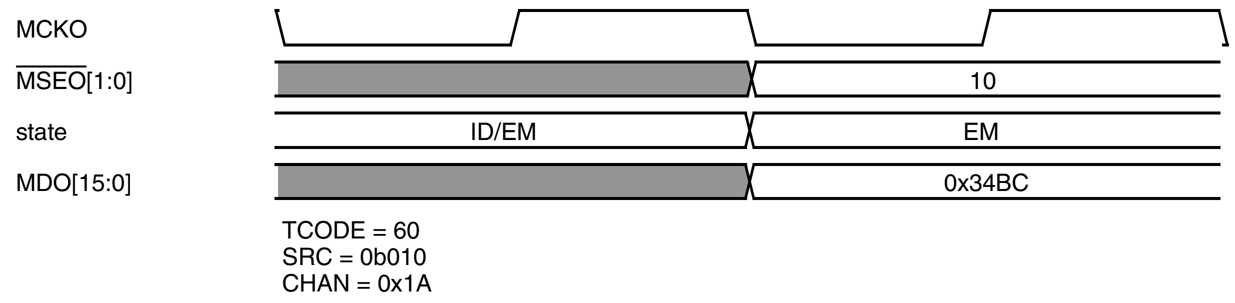


Figure 11-50. Channel Register Write Message

11.3.7 Data Trace

Dual engines perform loads and stores to the shared parameter RAM (SPRAM). The NDEDI block gathers information about these accesses on a dedicated bus. The NDEDI block traces all SPRAM accesses that meet the selected address range and attributes. This includes accesses from either engines as well as coherent dual-parameter controller (CDC) accesses. SPRAM accesses can have data sizes of 8, 24, and 32 bits. The NDEDI block supports all three data sizes.

Whenever performing 8-bits accesses, the eTPU accesses the 8 most significant bits of the word addressed, while the 24-bits accesses are accomplished by accessing the 24 least significant bits. Thus, since the eTPU system uses the Big Endian convention and since the eTPU data addresses are byte-relative the two least significant bits of the transmitted address follows the convention shown in Table 11-24.

Table 11-24. Two least significant bits for Data Trace Addresses

Value	Access
0b00	eTPU 8-bits or 32-bits accesses
0b01	eTPU 24-bits accesses
0b1x	Are not meaningful for eTPU

11.3.7.1 Data Trace Message Formats

There are four types of data trace messages (DTMs):

- Data write message
- Data read message
- Data write with synchronization message
- Data read with synchronization message

All of the data trace messages contain a SIZE packet. Table 11-25 shows the decoding for this packet.

Table 11-25. Data Trace SIZE Packet Decodings

SIZE Packet	Access Size
0b00	8-bits
0b01	24-bits
0b10	32-bits
0b11	Not Meaningful

11.3.7.1.1 Data Write Message

The data write message contains the data write value and the address of the target location relative to the address of the previous data trace message from the same source.

The data write message has the format shown in Figure 11-51.

[6 bits]	[K bits]	[2 bits]	[1-14 bits]	[1-32 bits]
TCODE (5)	SRC	SIZE	U-ADDR	DATA

Min Length = 10+K bits
Max Length = 52+K bits

Figure 11-51. Data Write Message Format

11.3.7.1.2 Data Read Message

The data read message contains the data read value and the address of the target location relative to the address of the previous data trace message from the same source.

The data read message has the format shown in Figure 11-52.

[6 bits]	[K bits]	[2 bits]	[1-14 bits]	[1-32 bits]
TCODE (6)	SRC	SIZE	U-ADDR	DATA

Min Length = 10+K bits
Max Length = 52+K bits

Figure 11-52. Data Read Message Format

11.3.7.1.3 Data Trace Synchronization Messages

The occurrence of any of the following conditions requires the following data trace message to be synchronizing:

- Exit of system reset
- Exit of low power mode
- Exit of debug mode
- Write to the development control register or data trace control register[1]
- Write to any of the eTPU data trace address range registers
- Forced end executed in eTPU engine
- Exit of NDEDI reset
- 255 data trace messages of the same source are queued without synchronization
- Assertion of the event in ($\overline{\text{EVTI}}$) pin, depending on how the development control register is configured
- A watchpoint occurrence[2]
- A queue overrun

Data trace synchronization messages provide the full addresses (without leading zeros) and ensure that the development tools fully synchronize with data trace regularly. Each synchronization message provides a reference address for subsequent DTMs, in which only the unique portion of the data trace address is transmitted.

[1]Only the source that have the register written will generate a synchronization message.
[2]A watchpoint occurrence for one engine generates synchronization message for that engine only.

There are two types of data trace synchronization messages: data write synchronization and data read synchronization.

Data Write Synchronization Message

The data write synchronization message has the format shown in Figure 11-53.

[6 bits]	[K bits]	[2-bits]	[1-14 bits]	[1-32 bits]
TCODE (13)	SRC	SIZE	F-ADDR	DATA

Min Length = 10+K bits
Max Length = 52+K bits

Figure 11-53. Data Write Synchronization Message Format

Data Read Synchronization Message

The data read synchronization message has the format shown in Figure 11-54.

[6 bits]	[K bits]	[2-bits]	[1-14 bits]	[1-32 bits]
TCODE (14)	SRC	SIZE	F-ADDR	DATA

Min Length = 10+K bits
Max Length = 52+K bits

Figure 11-54. Data Read Synchronization Message Format

11.3.7.2 Data Trace Operation

Data tracing is performed by snooping a dedicated eTPU/Nexus interface for SPRAM read and write cycles. Data trace functions are enabled by setting the appropriate fields in the following registers:

- ENGINE1 development control (NDEDI_ENGINE1_DC) register
- ENGINE2 development control (NDEDI_ENGINE2_DC) register
- ENGINE1 data trace control (NDEDI_ENGINE1_DTC) register
- ENGINE2 data trace control (NDEDI_ENGINE2_DTC) register
- CDC data trace control (NDEDI_CDC_DTC) register
- eTPU data trace address range 0 (NDEDI_ETPU_DTAR0) register
- eTPU data trace address range 1 (NDEDI_ETPU_DTAR1) register
- eTPU data trace address range 2 (NDEDI_ETPU_DTAR2) register
- eTPU data trace address range 3 (NDEDI_ETPU_DTAR3) register

For details on register configuration, refer to Section 11.2.1, “Register Descriptions.”

The dual engines and CDC data tracing share the same data trace address range registers. This enables usage of more address ranges for one client when the others are not being traced, or monitoring a certain range regardless of which client accesses the range.

The eTPU provides information to the NDEDI block when the access is actually being performed. The NDEDI does not have to track a bus protocol or check for error conditions on the accesses, since SPRAM does not generate errors. Any access signaled by the eTPU/NDEDI interface completes without error. Data trace flow is depicted in Figure 11-55.

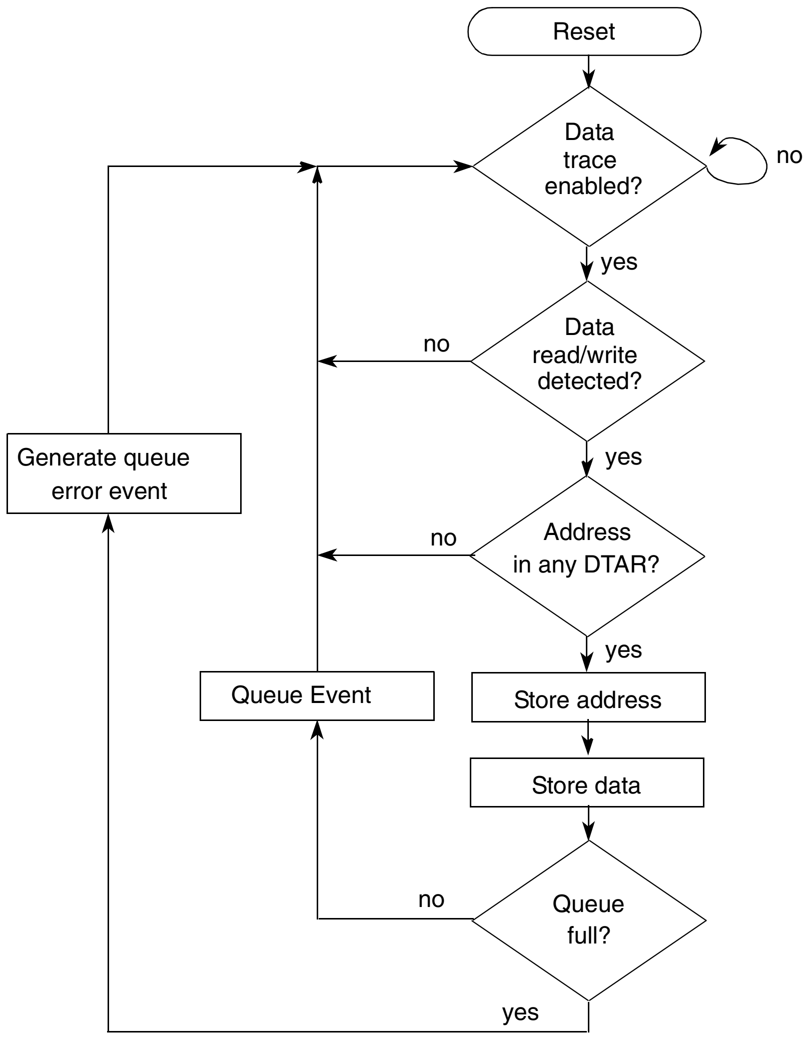


Figure 11-55. eTPU/CDC Data Trace Flow Diagram

11.3.7.2.1 Data Trace Windowing

Data trace windowing is provided so the development tool can decrease the auxiliary port usage by limiting the accesses that are traced.

Data trace windowing is achieved via the address range defined by the DTSA and the DTEA fields of the DTAR registers. These registers are shared by ENGINE1, ENGINE2, and the CDC. SPRAM accesses will be traced if their address fall in any of these ranges and the specific range is enabled in the data trace control register of the source making the access. Data read and/or data write trace may be enabled via the read/write trace control fields in the data trace control registers.

Data trace ranges are 32-bit aligned. This alignment is done by making the two least significant bits of the DTSA and DTEA fields read only. Since the two least significant bits of DTSA and DTEA are read only and default to different values, DTSA can never equal DTEA. Table 11-26 shows the relationship between the DTSA and DTEA fields.

Table 11-26. Data Trace Address Range Options

Programmed Value	Range Selected
DTSA < DTEA	DTSA –> <– DTEA
DTSA > DTEA	All addresses are out of Range [1]
DTSA==DTEA	Impossible [2]

[1] Since all addresses are considered to be out of Range, a Data Trace Event may be recognized if the DTC register is programmed so that addresses out of range are queued.

[2] DTSA can not be equal to DTEA since the two least significant bits of these field are always different. DTSA[1:0] = 00 and DTEA[1:0] = 11.

11.3.7.2.2 Relative Addressing

The relative address feature is compliant with the IEEE-ISTO 5001-2002 standard recommendations and is designed to reduce the number of bits transmitted for addresses of data read and data write messages.

The address transmitted for a determined source is relative to the address of the previous data trace message sent for that source. It is generated by XOR'ing the new address with the previous address and then using only the results up to the most significant '1' in the result. To recreate this address, an XOR of the (most-significant 0-padded) message address with the previously decoded address gives the current address. Figure 11-56 shows how a relative address is generated and how it can be used to recreate the original address.

```
Previous Address (A1) = 0xC10, New Address (A2) = 0xF64

    Address Generation:

    A1 = 1100 0001 0000
    A2 = 1111 0110 0100
A1 ^ A2 = 0011 0111 0100

    Address Message (M1) = 11 0111 0100


    Address Recreation:

    A1 ^ M1 = A2
    A1 = 1100 0001 0000
    M1 = 0011 0111 0100
A1 ^ M1 = 1111 0110 0100 = A2
```

$$A1 \wedge A2 = 0\underbrace{011\ 0111\ 0100}$$

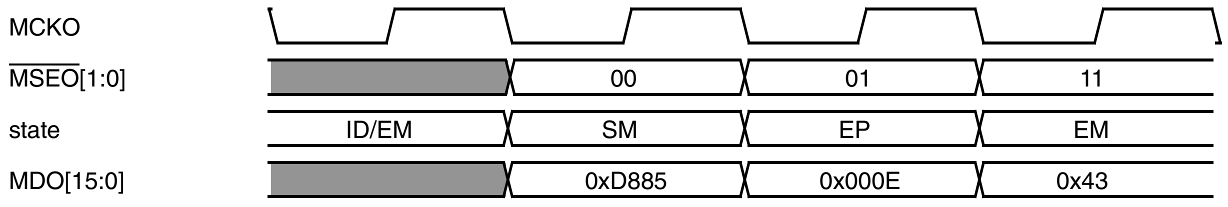
Figure 11-56. Relative Address Generation and Recreation

11.3.7.3 Timing Diagrams

Note that all of the following timing diagrams assume a 3-bit SRC size and either a 4-bit MDO port or a 16-bit MDO port. The state variable is not a signal, but instead is derived from $\overline{\text{MSEO}}$. It is included for clarity. Refer to Figure 11-20 for $\overline{\text{MSEO}}$ state diagram.

The following abbreviations are used for the state variable in the diagrams:

- ID = Idle
- SM = Start message
- NT = Normal transfer
- EP = End packet
- EM = End message



TCODE = 5
SRC = 0b010
SIZE = 0b00
U-ADDR = 0x1DB
DATA = 0x43

Figure 11-57. Data Write Message

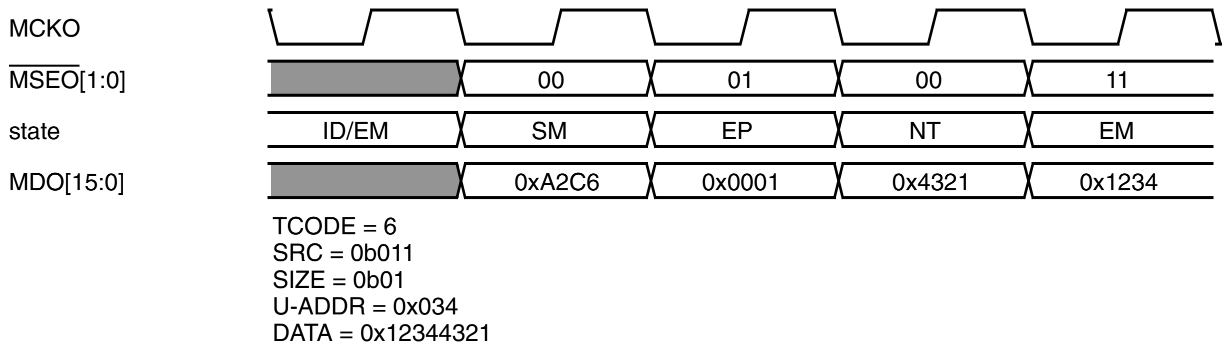


Figure 11-58. Data Read Message

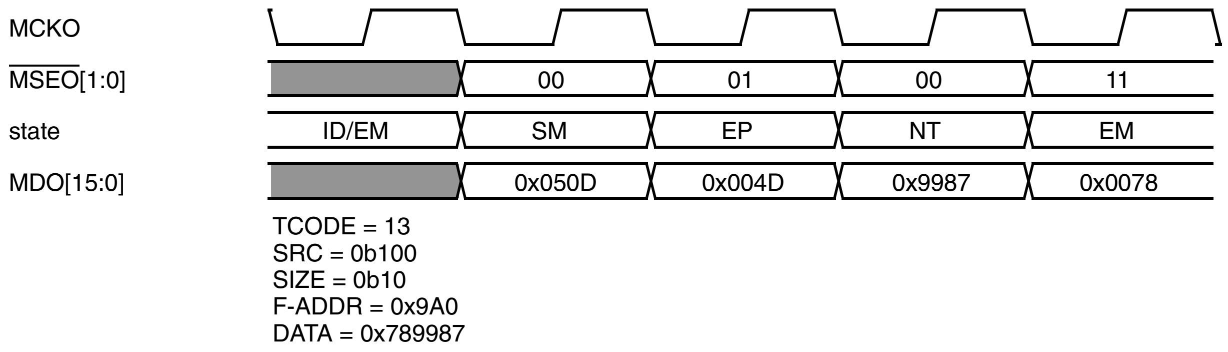


Figure 11-59. Data Write with Synchronization Message

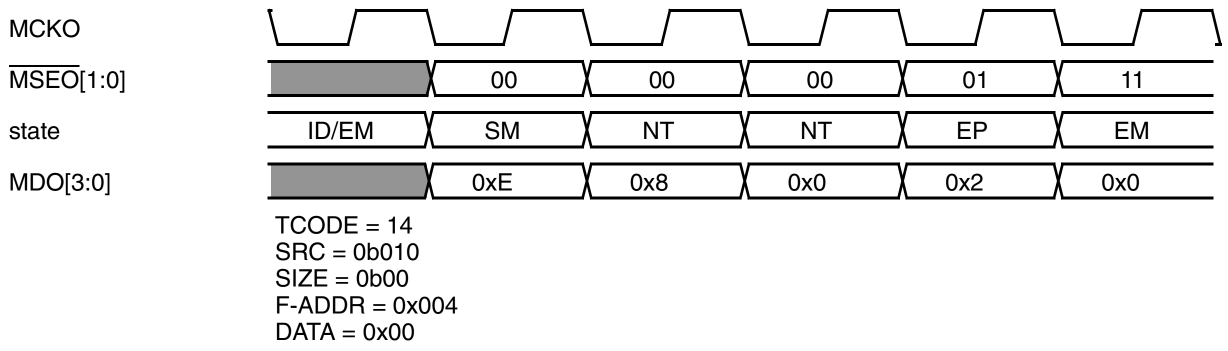


Figure 11-60. Data Read with Synchronization Message

11.3.8 Watchpoint Trace

The NDEDI module provides eTPU watchpoint messaging via the auxiliary port, as defined by the IEEE-ISTO 5001-2002 standard.

11.3.8.1 Messaging

The NDEDI block provides watchpoint messaging using IEEE-ISTO 5001-2002 standard-defined public messages. When a watchpoint occurs, a watchpoint event is sent to the event queue. If the watchpoint condition occurs while the event queue is not enabled for storing snapshots, a watchpoint overrun message is generated.

The watchpoint message has the format shown in Figure 11-61.

[6 bits]	[K bits]	[7 bits]
TCODE (15)	SRC	WPHIT

Length = 13+Kbits

Figure 11-61. Watchpoint Hit Message Format

The values for the WPHIT packet are described in Table 11-27. Notice that WPHIT is never 0 because the message is only generated when a watchpoint has occurred.

Table 11-27. WPHIT Values

Value	Description
0b1xxxxxx	eTPU watchpoint 1 (based on BWC1 register)
0bx1xxxxx	eTPU watchpoint 2 (based on BWC2 register)
0bxx1xxxx	Channel register write watchpoint
0bxxx1xxx	Host service request watchpoint
0bxxxx1xx	Link register watchpoint
0bxxxxx1x	MRL watchpoint
0bxxxxxx1	TDL watchpoint

11.3.8.2 Error Messages

A watchpoint overrun error event is queued if an watchpoint event occurs and the event queue is not enabled for storing snapshots. This event is queued as soon as the event queue becomes empty.

The error message has the format shown in Figure 11-62.

[6 bits]	[K bits]	[5 bits]
TCODE (8)	SRC	ECODE (0b00110)

Length = 11+K bits

Figure 11-62. Watchpoint Overrun Error Message Format

11.3.8.3 Synchronization

Upon the occurrence of a watchpoint, the next program and data trace messages are synchronization messages. A watchpoint in one of the engines causes data and program synchronization for that engine only.

11.3.8.4 Timing Diagrams

Note that all of the following timing diagrams assume a 3-bit SRC size and a 16-bit MDO port. The state variable is not a signal, but instead is derived from $\overline{\text{MSEO}}$. It is included for clarity. Refer to Figure 11-20 for $\overline{\text{MSEO}}$ state diagram.

The following abbreviations are used for the state variable in the diagrams:

- ID = Idle
- SM = Start Message
- NT = Normal Transfer
- EP = End Packet
- EM = End Message

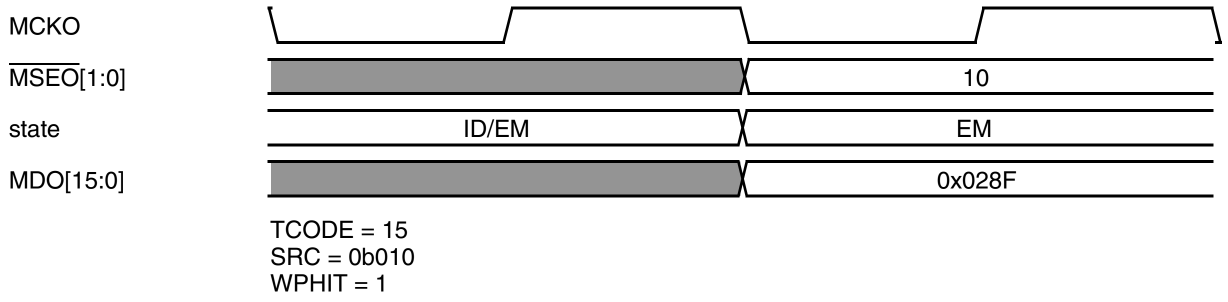


Figure 11-63. Watchpoint Message

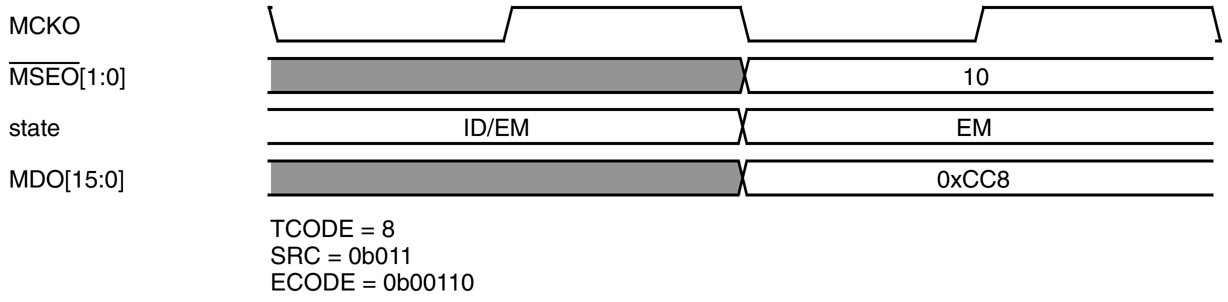


Figure 11-64. Watchpoint Overrun Error Message

11.3.9 eTPU Message Queue

The NDEDI implements an external Q_SIZE location queue that is shared by data trace, ownership trace, program trace, watchpoint, error and debug status events.

One single microcycle may generate a combination of data trace, ownership trace, program trace, watchpoint, error and debug status events. The data trace events related to one engine have different timing though, being generated at the system clock subsequent to the other events. Thus, since the ENGINE1 and ENGINE2 alternate instructions, a snapshot may contain data trace information regarding to one engine and other events information related to the other engine.

With the occurrence of one or more events at one specific cycle, a snapshot of the signals, that can give information about all events, is queued in only one location of the queue. The information of the source which have generated the events is also queued at the same position, so the message formatter can reconstruct the messages of the occurred events. Figure 11-65 shows the event queue block diagram.

The queue operation is transparent to the user.

The event queue is read whenever there is at least one position of the FIFO occupied and there are no concurrent writes to the queue. Thus, the appropriate message is formatted and a request to send the message is sent to the Nexus Port Controller (NPC). The request will be assigned a level based on the number of used positions in the Queue as shown in Table 11-28.

Table 11-28. MDO Request Level

Level	Used positions (q) as a fraction Queue size
0	q = 0
1	1 ≤ q < 0.5
2	0.5 ≤ q < 0.75
3	0.75 < q

According to the request priority of the NDEDI and the other Nexus modules within the chip, the NPC will grant the Auxiliary port to the NDEDI, and as soon as the auxiliary port is granted, the NDEDI will send the message. If there are other pending messages in the same snapshot, a new request is sent and a new grant is expected from the NPC. The next snapshot is only read when there are no pending messages in the event queue and there are no concurrent write accesses to the Queue.

If more than one message event are to be sent, the message formatter sends first the messages related to debug status, then the messages related to watchpoints, then the messages related to program trace, and last the data trace messages. If the Q_SIZE position queue is full and an event occurs, the event queue is disabled for storing any information. When the queue gets empty, an error event is queued containing information of all lost events. At this point, snapshots may be queued again.[1]

NDEDI maintain temporal ordering of messages relative to ENGINE1, ENGINE2 and CDC, but not relative to messages generated by other clients sharing the auxiliary port.

[1]If ENGINE*n* is in stall condition, and an error happens due to the other engine or to the CDC operation, the ENGINE*n* operation will only be resumed after the queue is empty.

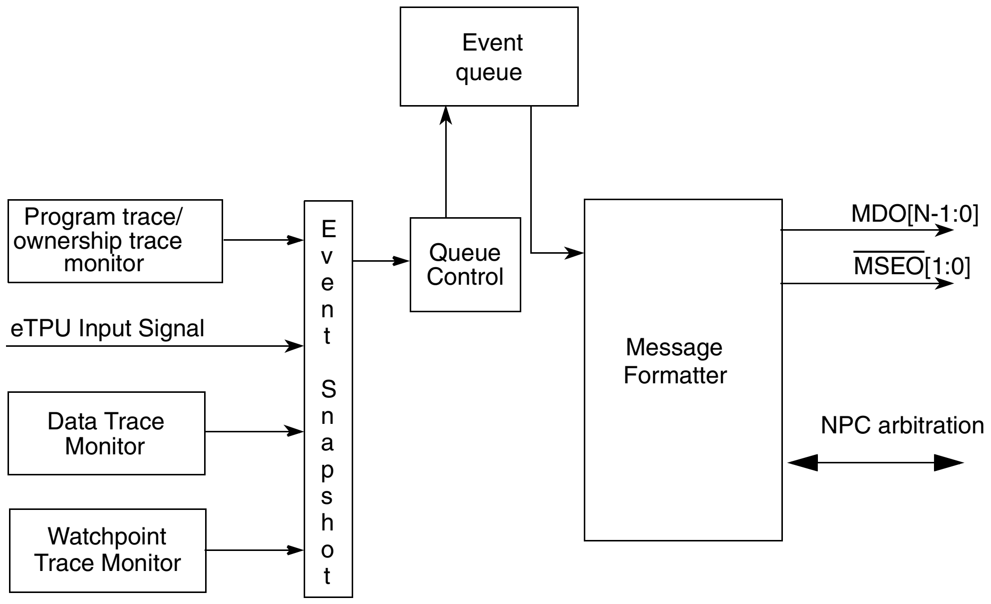


Figure 11-65. Event Queue

11.3.9.1 Queue Control

The queue is able to perform a single read or write operation per system clock. Thus, since there are no buffers to neither traces nor watchpoint events, the message formatter can only read the event queue if no events happen at the same time.

11.3.9.2 Error Messages

A trace or watchpoint overrun error occurs when an event cannot be queued. The error code (ECODE) within the error message indicates what type of trace overrun has occurred and that messages might have been lost for any or all of these events. Table 11-20 shows the error code encodings for the various overrun error conditions.

When information is lost, the event queue is disabled and no events are queued until the event queue gets empty. After the event queue gets empty an error message is queued with the information of which information was lost for each engine. At this point the queue is enabled for storing snapshots again.

The error message indicates which source lost information and what kind of trace information was lost. If multiple sources lose information at the same time multiple error messages are sent.

The error message has the format shown in Figure 11-66.

[6 bits]	[K bits]	[5 bits]
TCODE (8)	SRC	ECODE

Length = 11+K bits

Figure 11-66. Error Message Format

11.3.9.3 Timing Diagrams

Note that the following timing diagram assumes a 3-bit SRC size and a 16-bit MDO port. The state variable is not a signal, but instead is derived from $\overline{\text{MSEO}}$. It is included for clarity. Refer to Figure 11-20 for $\overline{\text{MSEO}}$ state diagram.

The following abbreviations are the values for the state variable in the diagram:

- ID = Idle
- SM = Start message
- NT = Normal transfer
- EP = End packet
- EM = End message

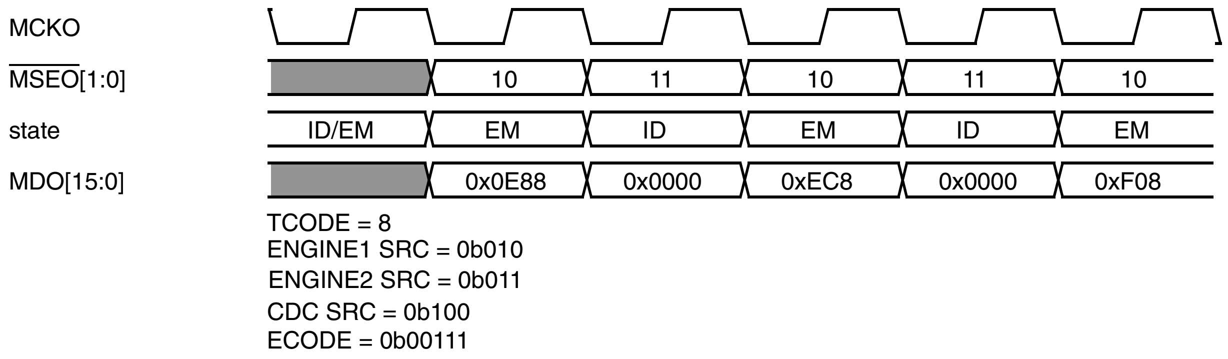


Figure 11-67. Error Messages (Program/Data/Ownership Trace Overrun)

Notice that three overrun messages are transmitted in Figure 11-67. The first is an overrun indication for ENGINE1, the second is an overrun indication for ENGINE2, and the third is an overrun indication for the CDC.

11.4 Initialization/Application Information

11.4.1 Accessing NDEDI Tool-Mapped Registers

To initialize the JTAG port for register accesses, the following sequence is required:

1. Enable the Nexus TAP controller
2. Retrieve device ID if needed
3. Load the Nexus TAP controller with the NEXUS-ENABLE instruction

To write control data to NDEDI tool-mapped registers, the following sequence is required:

1. Write to the client select control register, if needed
2. Write the 7-bit register index and set the write bit to select register with a pass through the SELECT-DR-SCAN path in the JTAG state machine
3. Write the register value with a pass through the SELECT-DR-SCAN path. Notice that the prior value of this register is shifted out during the write

To read status and control data from NDEDI tool-mapped registers, the following sequence is required:

1. Write to the client select control register if needed
2. Write the 7-bit register index and clear write bit to select register with a pass through SELECT-DR-SCAN path
3. Read the register value with a pass through the SELECT-DR-SCAN path. Data shifted in is ignored

11.4.2 Program Trace Reconstruction

11.4.3 Microcode Development Support

The following sections describe the steps required to perform various class 1 features with the NDEDI block.

11.4.3.1 Read and Write SPRAM In Debug Mode

Reading and writing to the SPRAM is done through Nexus read/write access hardware external to the NDEDI module. SPRAM reads and writes are done the same in debug mode as they are in run mode.

11.4.3.2 Read and Write eTPU Internal Registers in Debug Mode

1. Select an area of the SPRAM to be used for dumping the register values
2. Read the current value of those SPRAM locations and store them
3. Write to the SPRAM as needed for writing internal eTPU registers
4. Execute external microcode that reads values from the SPRAM location and writes the values to registers or reads registers and writes the values to the SPRAM. See Section 11.4.3.8, “Execute Forced Microcode Instruction in Debug Mode,” for more details
5. Read the SPRAM as needed to retrieve register read values
6. Write the SPRAM back to the original values

11.4.3.3 Enter Debug Mode at the Negation of Reset

To enter debug mode at the negation of reset, configure the eTPU’s development control register during reset so that the DBE is set. Configure other bits as appropriate.

11.4.3.4 Enter Debug Mode During Normal Execution

To enter debug mode during normal execution, write the eTPU’s development control register to set the DBE and DBR bits. Configure other bits as appropriate.

11.4.3.5 Stop Program Execution on a Breakpoint

There are four possibilities to stop the program execution on a Breakpoint:

1. Program execution stops at the end of the current microcycle when a synchronous breakpoint condition is reached. A synchronous breakpoint condition occurs due to a twin engine breakpoint, an external source (ipg_debug) request (depending on the value of the CBT field at the DC register) or due to certain conditions at the beginning of a thread (refer to Section 11.2.1.10, “ENGINEn Breakpoint/Watchpoint Control 3 Register (NDEDI_ENGINEn_BWC3) for more information”).
2. Program execution stops at the beginning of the current microcycle when an asynchronous breakpoint condition is reached. An asynchronous condition occurs due to a SPRAM access breakpoint, an illegal instruction breakpoint, or at the first microinstruction after a write operation to the CHAN register
3. Program execution stops at the end of the current thread if a twin breakpoint or an external source (ipg_debug) requests to enter in debug mode and the NDEDI is programmed to enter in debug mode only at the end of the current thread
4. Whenever fetching an instruction, if the microprogram counter matches the BWA related to one of the breakpoint conditions programmed for breaking into an instruction fetch, the instruction is tagged indicating that it should not be executed. Thus, the microengine halts just before the execution of this instruction if this instruction should be executed. So, it won’t stop in case the tagged instruction is the instruction following a Jump with Flush instruction

Refer to Section 11.4.3.7, “Set Breakpoint or Watchpoints,” and Section 11.2.1, “Register Descriptions,” for details on configuring breakpoints.

11.4.3.6 Single Step Instructions and Re-Enter Debug Mode

To single step an instruction from shared code memory in debug mode, write to the engine’s development control register so the CBR bit is asserted, the DBR bit is cleared and the SS bit is set. Single stepping this way causes the Engine to exit debug mode for a microcycle where the TCRs, decrementors, and microprogram counter can update. Program Trace events are not generated within Single Step operation.

11.4.3.7 Set Breakpoint or Watchpoints

There are two main eTPU breakpoint/watchpoint sources that are independent of each other but have identical configuration registers. Each can be configured as either a breakpoint or watchpoint that detects reads and/or writes, compares data and/or address. Each can be configured to occur only on certain channel services.

In addition, there is a breakpoint/watchpoint that can be configured to occur when the CHAN register is written to a certain value.

There are also four channel service breakpoint/watchpoints that can be configured to occur when a specific channel is serviced for one of the following reasons: host service request, link request, match detect, and transition detect.

Breakpoints cause the Engine to halt. Watchpoints cause a watchpoint hit message. All breakpoint/watchpoint sources can be configured to assert $\overline{\text{EVTO}}$ on an breakpoint/watchpoint occurrence.

Refer to the applicable implementation-specific reference manual for detailed descriptions of breakpoint/watchpoint registers.

11.4.3.8 Execute Forced Microcode Instruction in Debug Mode

To execute a forced instruction in debug mode, write the instruction into the microinstruction debug register. This mechanism does not cause the engine to exit debug mode so the microprogram counter is not updated (unless a branch instruction is executed) and the TCRs and decrementors are not updated if the CLKS bit is asserted in the development control register. Executing instructions in this manner is useful for reading and writing internal engine registers.

Chapter 12 Initialization/Application Information

12.1 Configuration Sequence

After initial power-on reset the eTPU remains in an idle state, requiring initialization of several registers before any function can begin execution. Also, if the SCM is implemented in RAM, it should be initialized with the eTPU application code prior to configuring the ETPU. Configuration procedures are summarized as follows:

- If SCM is implemented as RAM, load the eTPU application code.
- Initialize the SCM MISC logic (see Section 10.3.1, “SCM Test for MISC (Multiple Input Signature Calculator)”).
- Initialize the eTPU time base configuration registers (ETPUTBCR) to setup:
    - TCR1 and TCR2 prescalers and clock sources.
    - Select digital filtering mode.
    - TCRCLK signal filter control.
    - Angle mode operation (if necessary).
- Initialize the eTPU engine configuration register (ETPUECR) to setup:
    - Entry table base.
    - Filter prescaler clock control.
- Initialize eTPU configuration register (ETPUREDCR) to setup TCR1/2 resource client/server operation.
- Write to the Channel Configuration registers (ETPUCxCR) to choose the function to be performed by each channel, and its parameter base address.
- Write to channel status control register (ETPUCxSCR) to choose the possible variations within the function flow (FM bits).
- Write to SPRAM for parameter initialization of each configured channel.
- Write to channel x Host Service Request registers (ETPUCxHSRR) to initialize the active channels.[1]

[1] This operation is done before enabling active channels to avoid time events happening before the channel initialization.

- Write to the channel interrupt enable register (ETPUCIER) if interrupts are to be enabled from the appropriate channels. Likewise for Data Transfer Requests (ETPUCDTRER). This can be done through ETPUCxCR.
- Write to channel x configuration registers (ETPUCxCR) to enable each channel by assigning it a high, middle, or low priority (CPR field).[1]
- Monitor the host service request registers (ETPUCxHSRR) for completion of initialization.
- Write ETPUMCR[GTBE] = 1 to start TCR1/TCR2 time base counting at same time in both Engines.

See Appendix B, “eTPU Initialization Code Example.”

12.2 Reset Options

12.2.1 Hardware Reset

Both engines and common logic are reset, and even the System Configuration and Global Channel registers assume their reset values.

12.2.2 Software Reset

The eTPU has no Software reset. It only has the Force END mechanism to break (suspected) infinite loops if they appear to have been entered. For more information, see field FEND in Section 4.1.4, “eTPU Engine Configuration Register (ETPUECR).”

12.3 Multiple Parameter Coherency Methods

This section contains descriptions of two methods for coherent transfer of multiple parameters between the host and eTPU. Both methods involve the use of two parameter areas: the transfer parameter area (hereafter called TPA), which is the SPRAM area directly accessed by the host for reads and writes, and the permanent parameter area (hereafter called PPA), which are the SPRAM positions where channel parameters are normally accessed by function microcode. Note that parameters in either TPA or PPA do not have to be in sequential addresses. TPAs and PPAs allocation are completely defined by the application, and there may be any number of them, independent of the channels.

The methods described here are not the only solutions for the coherent transfer problem, and both can coexist in eTPU and even used in combination. Also note that for transfers of a pair of parameters, the coherent dual-parameter controller is faster and has less impact on both eTPU and host performance. That being said, the two methods are:

- Transfer Service: upon host service request, a microengine thread transfers data from/to a TPA to/from a PPA.Coherency is guaranteed by the fact that a thread is atomic with respect to other threads in the same engine, and so are its transfers. If parameters in PPA are shared by both engines, hardware semaphores have to be used to access them.

- Mailbox: for host to eTPU transfers, the microcode checks a flag set by the host, which indicates the existence of new parameter data in the TPA. The microcode can then either access TPA data directly or copy it to the PPA. For eTPU to host transfers, when microcode changes PPA, it copies them to the TPA and flags updated TPA data to host using an interrupt or a data transfer request. The mailbox flag is reset when data is copied by the eTPU microcode, when it transfers TPA to PPA (possibly followed by an interrupt), **or** by the host, when it reads data from the TPA. This indicates that TPA is free for another transfer.

Transfer service has the advantage of separating the task of data transfer from the functional service thread that accesses the parameters, with less impact to the latter. However, compared to the mailbox method, the transfer service method has a longer average latency, because the transfer service thread has to contend for a time slot to execute. This latency can be minimized if transfer service thread is assigned to a separate channel with higher priority, but even this does not guarantee that PPA is updated before the next execution of the functional thread that uses it.

The mailbox method, on the other hand, makes the functional thread check for the existence of new data (host to ETPU). It does not have to be responsible for the transfer. The mailbox method may access the TPA directly, and once the access is finished, a transfer service can be used to copy data from TPA to PPA.

12.4 Programming Hints and Caveats

12.4.1 Atomic Dual Access After a Call, Return

A dual, back-to-back parameter access is not atomic after a call, a jump, or a return if they occur in parallel with an odd SPRAM access. It is safer to make a pair of parameter accesses that must be coherent begin at the second instruction after a call/jump/return.

12.4.2 Resource Polling

The use of polling while waiting for a condition or a resource (except semaphore lock) should be avoided in order not to hang the processor in long loops. This general programming guideline is greatly enforced in eTPU microengine, as a thread cannot be suspended for any reason. Safer polling, albeit with long and indeterministic latency, can be obtained if one issues a channel link to itself and terminates the thread. The microengine is then free to do other tasks, and the next poll happens at the next time the channel is serviced. This mechanism can be combined with finite (timed out) loops for better latency.

12.4.3 Changing Channel Function, Parameter Base, or Entry Table Scheme

Channel Function, Parameter Base Address and Entry Table Scheme are determined by the ETPUCxCR register fields CFS, CPBA and ETCS. They cannot be changed when the channel is enabled. If the channel is disabled first, one may still have service requests from the previous function, so before the channel is enabled again one must be sure that:

- the first thread executed in the new function is the initialization one.
- the initialization thread of the new function clears any previously pending service request.

Follow a safe procedure for function changing:

1. disable the channel (write ETPUCxCR field CPR=00).
2. change the function configuration (ETPUCxCR fields CFS and/or CPBA and/or ETCS).
3. request the initialization thread, writing ETPUCxHSRR with the initialization HSR (channel still disabled).
4. enable the channel (write ETPUCxCR field CPR > 0); the initialization HSR is serviced before any other formerly pending service requests, clearing them.

12.4.4 Checking and Clearing Interrupts of a Stopped Engine

An engine may be stopped with interrupts (or DMA requests) pending. This includes the case when the engine's MDIS bit is set and a thread is still running: the thread will complete execution, possibly issuing an interrupt or DMA request before the engine stops, setting the STF bit.

As soon as the engine stops the channel registers become inaccessible, issuing bus errors when accessed. However, interrupts and DMA requests can still be checked and cleared through the Global Channel Registers. DMA requests can also be cleared by the hardware handshaking with the DMA controller when the engine is stopped.

12.5 Estimating Worst Case Latency

Reliable systems are designed to work under worst-case conditions. This section explains how to estimate worst-case latency (WCL) for any eTPU function in any system. This section covers the following topics:

- Introduction to worst-case latency
- Using worst-case latency estimates to evaluate performance
- Priority scheme details used in WCL analyses
- First-pass WCL analysis

- Second-pass WCL analysis

The first-pass WCL analysis is based on a deterministic, generalized formula that is easy to apply. Because of the generalizations in the formula, the first analysis result is almost always much worse than the real worst case. If the desired system performance is within the limits of this first analysis, then no further analysis is required; the system is well within the performance limits of the eTPU. If the desired system performance exceeds that indicated by the first analysis, the second-pass WCL analysis should be applied. The second-pass analysis is not a generalized formula, but rather uses specific system details for a realistic worst-case estimation.

12.5.1 Introduction to Worst-Case Latency

NOTE

In this section the latency calculation and examples refer to old TPU functions such as PWM, DIO etc. These functions use single action channels which have single transition and single match functionality. They are not optimized for the eTPU hardware enhancement which support various double action modes. These examples are for reference only. New eTPU functions which are optimized for the new hardware will impose different latency calculations.

Worst-case latency for a channel is the longest amount of time that can elapse between the execution of any two function states on that channel. For example, if in a particular system, channel 5 is running PWM, the worst-case latency for channel 5 is the longest possible time between the execution of two PWM threads. The worst case time includes the time the execution unit takes to execute threads for other active channels, and other delays described later in this section. Refer to Figure 12-1.

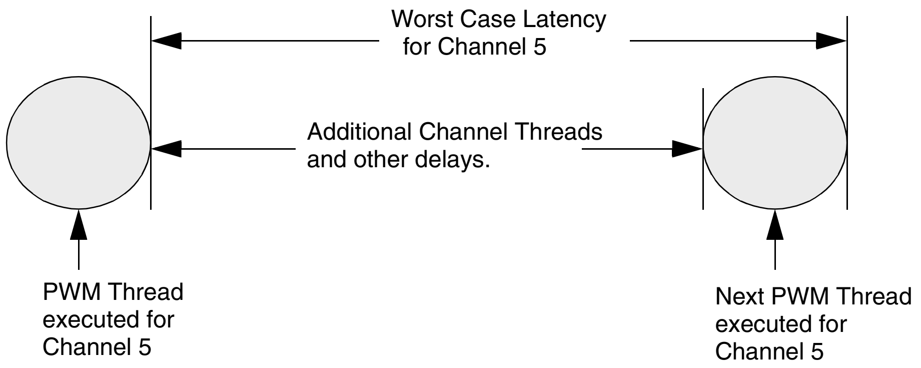


Figure 12-1. Worst-Case Latency for PWM

Worst-case latency for a channel depends both on the function running on that channel and on the activity on other channels. Since the 32 eTPU channels must all share the same execution unit, execution speed of a particular function varies with each system. The PWM thread response is faster if there are no other active channels than if other channels are also active. In addition, changing the priority scheme and channel number assignments can change performance for a function even if the same set of functions are still active.

Each function is divided into threads, as shown in Figure 12-2, see also Section 5.1, “Functions and Threads.” The eTPU microengine executes one thread of a function at a time. For example, the microengine might execute thread 1 of PWM, then thread 3 of DIO, then thread 2 of PWM, then thread 2 of SM, and so on. The amount of time the eTPU microengine grants a function to execute a thread varies with the number of microcode instructions in the thread.

Since there is only one eTPU microengine (in each eTPU engine), the eTPU cannot actually execute the software for multiple functions simultaneously. However, the hardware for each of the channels is independent. This means that, for example, all 32 channel signals can change thread at the same moment, provided that the function software sets up the channel hardware to do so beforehand.

With host CPU code, the system designer assigns functions to channels and initializes the functions. After initialization, functions typically run without host intervention, except for eTPU channel interrupts to the host to give or receive information. Most functions can run continuously with periodic servicing from the eTPU microengine. As required, the channels request service from the eTPU microengine and the eTPU scheduler determines the order in which the channels are serviced. Worst-case latency for a channel can be derived from the details of the priority scheme that the scheduler uses. For scheduler details, see Section 5.3, “Scheduler.”

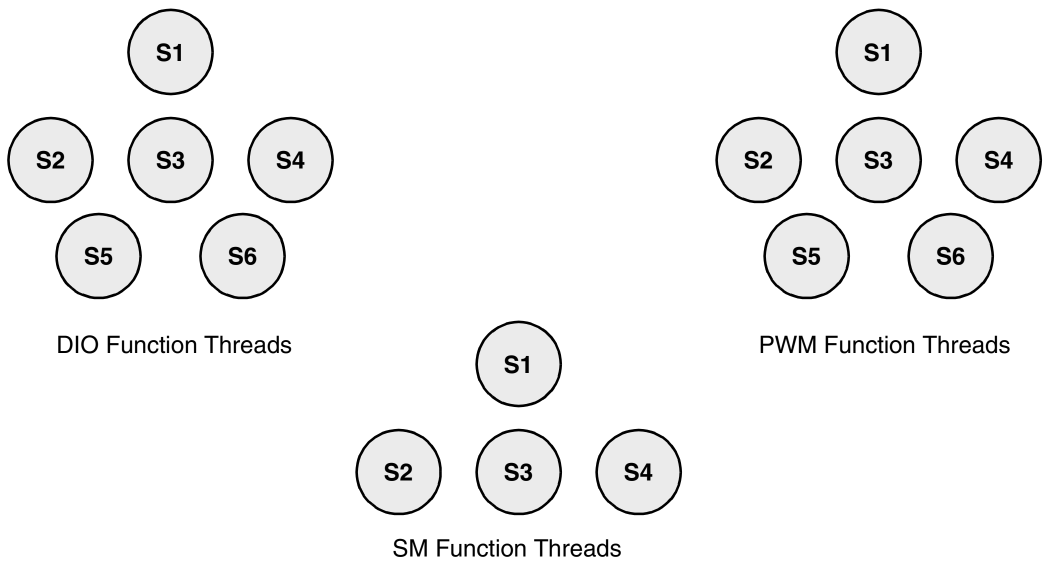


Figure 12-2. Function Threads

12.5.2 Using Worst-Case Latency Estimates to Evaluate Performance

Once the WCL is found for a channel, the user must determine how to use this number to analyze performance. To analyze the performance of a channel running the PWM function, for example, some information about what happens in each thread is necessary.

The following example refers to an old TPU PWM function, which is not optimized to the eTPU enhanced hardware. For the PWM, thread 1 is the initialization thread, and threads 2 and 3 are used during normal function execution. PWM threads 4, 5, and 6 are for special modes and will be assumed to be unused on channel 5.

Thread 2 writes a time into the channel 5 match register and performs other operations that will cause the channel 5 signal to go from low to high at the time indicated in the match register (match time). At match time, the signal goes high and channel 5 requests service from the eTPU microengine to execute thread 3. Thread 3 writes a time into the channel 5 match register and performs other operations that will cause the channel 5 signal to go from high to low at match time. At match time, the signal goes low and channel 5 requests service from the eTPU microengine to execute thread 2. A PWM wave is kept running on the system by the eTPU executing thread 2, then thread 3, then thread 2, then thread 3, and so on.

Since the definition of worse-case latency assumes a fully loaded running system, initialization threads are not part of worst-case calculations. For the channel 5 example, the two PWM threads in Figure 12-1 are thus the two normal running threads, threads 2 and 3.

Figure 12-1 does not define which thread is thread 2 and which is thread 3. Since the worst-case latency derived from the first-pass analysis is the worst case between any 2 threads (not counting initialization threads), it is safe to say that the worst-case latency shown in Figure 12-2 represents both the worst-case high time and the worst-case low time.

Notice in Figure 12-1 that worst-case latency is drawn from the end of the execution of the first PWM thread to the end of the execution of the next PWM thread. It is drawn from end to end because the microcode instructions that make up the threads control the channel hardware. To make sure that all the microcode instructions needed to change the pin thread have been executed, it is necessary to include the execution time of the second thread.

12.5.3 Priority Scheme Details Used in WCL Analysis

The user assigns functions to channel numbers and gives each active channel a priority level of high, middle, or low. The scheduler uses the channel number and channel priority level to determine the order in which to grant service.

The scheduler allocates time slots to specific priority levels of high, middle, or low. One function state is executed in each time slot. The length of a time slot varies according to the length of the executing state. When fully loaded, the scheduler always assigns time slots in a seven-slot sequence, see Figure 12-3. After a seven-slot sequence is completed, another seven-slot sequence begins, see Figure 12-4. Note that in the eTPU, when no service request exists, the scheduler goes to thread 1, but the WCL calculation considers the full load.

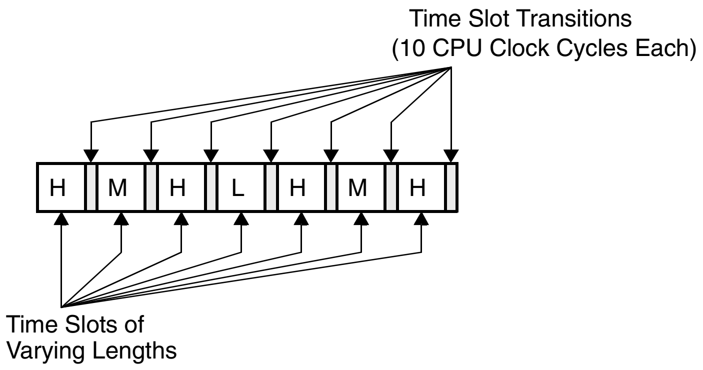


Figure 12-3. Time-Slot Sequence

This sequence scheme gives higher-priority channels more service time than lower-priority channels. High-priority channels are allocated four of seven time slots, middle-priority channels are allocated two of seven time slots, and low-priority channels are allocated one of seven time slots.

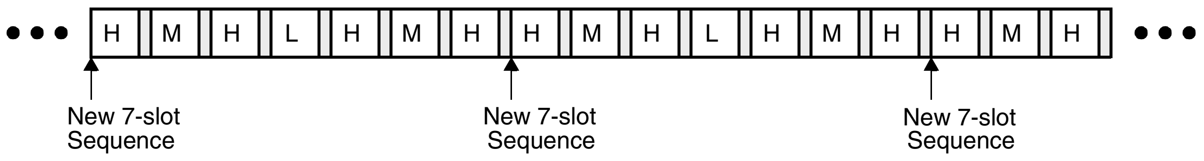


Figure 12-4. Multiple Time-Slot Sequences

12.5.3.1 Priority Passing

If no channel of the priority level assigned to the time slot is requesting service, the eTPU scheduler can pass priority to other levels. If no high-level channel is requesting service during a high level time slot, a middle-level channel is granted service; or, if no middle level-channel is requesting service, a low-level channel is granted service. If no middle-level channel is requesting service during a middle-level time slot, a high-level channel is granted service; or, if no high-level channel is requesting service, a low-level channel is granted service. If no low-level channel is requesting service during a low-level time slot, a high-level channel is granted service; or, if no high-level channel is requesting service, a middle-level channel is granted service. If no channel is requesting service, the time slot sequence is reset to state 1 and the scheduler idles until a request is received.

Priority passing is implemented in hardware and does not contribute to worst case latency.

12.5.3.2 Time-Slot Transition

After each time slot, the eTPU must prepare for the next time slot. This preparation time between each time slot is called a time-slot transition. See Section 5.1.2, "Time Slot Transition." Time-slot transitions take from 6 to 10 system clocks.

12.5.3.3 Channel Number Priority

If more than one channel of a priority level is requesting service, the lowest numbered channel is granted service first. For example, if channels 0, 5, and 9 are all high-level channels requesting service during a high time slot, channel 0 is granted service first. Continuing this example, if channel 0 requests service again immediately after being serviced, it is not serviced again until channels 5 and 9 are serviced. This scheme is implemented so that continuously-requesting low numbered channels do not take all the time on the eTPU execution unit and leave no time for other channels.

The scheduler uses registers to keep track of which channels have been serviced and which require servicing. Each channel has two register bits: a service request register (SRR) and a service grant register (SGR). The SRR is set when a channel requests service. After the channel has been granted service, the SGR is set and the SRR is cleared.

SGRs are not cleared individually by channel, but rather as priority level groups. The clearing of a group of SGRs begins a new cycle for that priority level. An SGR group is cleared on the condition that a channel of that priority level has just been serviced, and no other channel of that priority level is requesting service (has a set SRR) and has not been granted service (has a clear SGR).

For example, if a middle-priority channel has just been serviced (either in a middle-priority time slot or a high or low-priority time slot gained by priority passing), the SRRs and SGRs of all middle-priority channels are compared. If there is no middle-priority channel with its SRR set and SGR cleared, the scheduler clears all middle-level SGRs. If there is a middle-level channel with its SRR set and SGR cleared, the scheduler does not clear the SGR group, and the requesting middle-level channel is serviced on the next middle-level time slot (or possibly sooner by priority passing).

12.5.3.4 SPRAM Collision Rate

Most function threads read or write to the eTPU SPRAM at least once. Because both the eTPU microengine and the host can access the SPRAM but not at the same time, the microengine may suspend execution during the SPRAM access while waiting for the host to finish accessing the SPRAM. At other times the host may wait for the microengine. Wait states can take up to two system clocks, when the host accesses the SPRAM directly, without using CDC. Microengine wait-states must be added into the worst-case latency calculation. The system designer should estimate the percentage of SPRAM accesses in the system that will result in microengine wait-states. This percentage is called the RAM collision rate (RCR). In each collision with direct host accesses to the SPRAM, the microengine waits for two system clocks.

In the eTPU the coherent dual-parameter controller (CDC) may also access the SPRAM for atomic transfers of two parameters. The eTPU microengine may wait on this operation (if it is in service time) until the transfer is complete. The CDC always transfers two parameters, making four consecutive accesses (read, write, read, write) of one system clock each. The system designer should estimate the percentage of SPRAM accesses in the system that will result in a microengine wait due to coherent transfer, and multiply it with the average number of system clocks the microengine waits for each transfer. This percentage is called the coherent parameter collision rate (CPCR).

In addition, microengine-to-microengine multiple parameter coherent communication, using the hardware semaphores, may hold one microengine which is waiting to lock the semaphore while the other microengine is holding it. This waiting is due to a software loop, not hardware wait-states. Note that single parameter access of one microengine does not affect the timing of the other microengine due to SPRAM time interlace. This implies that single parameter microengine-to-microengine communication does not affect the performance. The microengine which is waiting for the semaphore will loop until it is freed by the other microengine. This time depends on the eTPU application. The system designer should estimate the percentage of microengine-to-microengine coherent multiple parameter coherent communication that will result in the eTPU, and multiply it with the average number of system clocks the microengine is stalled for each such transfer. This percentage is called CCR (communication collision rate).

A 100% collision rate for a system is the theoretical worst case. In many systems, however, the RCR, CPCR, and CCR would be very low, sometimes even near 0%. This is because the eTPU is an independent processor capable of servicing most function needs. Thus, the host rarely needs to access the eTPU parameter RAM. Also coherent microengine-to-microengine communication of more than one parameter may be rare. To find a realistic RCR and CPCR, the system designer should evaluate the host code and find the percentage of time it accesses the eTPU parameter RAM with or without using the CDC. This percentage gives a good RCR and CPCR. The eTPU application provides a good estimation of CCR.

NOTE

The programming practice of polling a flag in the eTPU SPRAM causes a very high RCR and should be avoided in high-performance systems.

After the collision rate for a system is found, it can be applied to the WCL calculations for each channel. The system designer can use the collision percentage and the number of

SPRAM accesses (with and without semaphores) to estimate the eTPU loop time for a function. Note that in old TPU functions CPCR and CCR are both zero.

The estimation of eTPU stall time is as follows:

Variables:

- N1 = number of simple RAM accesses in the longest thread
- RCRWait = maximum system clock stall time for simple RAM collision = 2
- CPCRWait = average system clocks for coherent parameter transfer (using CDC).
- N2 = number of eTPU-eTPU semaphore RAM accesses in the longest thread
- CCRWait = average system clocks for microengine-microengine communication transfer.

Estimated stall time:

```
Function eTPU maximal wait time =
N1 * (RCR * RCRWait + CPCR * CPCRWait) + N2 * CCR * CCRWait
```

12.5.4 First-Pass Worst-Case Latency Analysis

Following is the first-pass calculation of worst-case latency for a channel. Remember that this analysis uses generalizations that usually produce a result much worse than the real worst case. If the worst-case result from the first analysis is too long for the desired performance, use the second analysis for a more realistic worst-case analysis.

12.5.4.1 Worst-Case Assumptions and Formula

To estimate worst-case latency for a channel, assume this worst-case condition: the channel has just been serviced in a time slot of its priority level, and all other channels in the system are continuously requesting service and have cleared SGRs. The worst-case latency is the time from the end of the channel's service until the end of the channel's next service. See Figure 12-5.

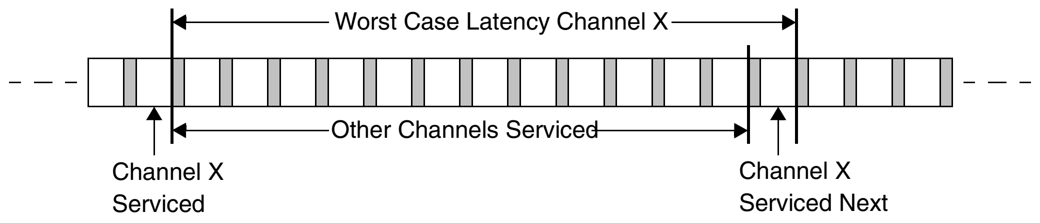


Figure 12-5. First-Pass Worst-Case Latency

To estimate worst-case latency:

- Find the worst-case service time for each active channel.

- Using the H-M-H-L-H-M-H time-slot sequence, map the channels that are granted for each time slot.
- Add time for six-clock time-slot transitions.

12.5.4.2 Finding the Worst-Case Service Time for Each Active Channel

A table for eTPU functions should lists the longest threads (not counting initialization threads) for the functions, and the number of eTPU SPRAM accesses in the longest state (semaphored and non-semaphored). These figures will be used for estimating microengine wait time.Table 12-1 is an example for old TPU functions in which there are only simple parameter RAM accesses. It does not take into consideration the CDC operation and microengine-to-microengine communication.

The worst-case service time for each channel is: (CPCR=CCR=0)

Longest thread + (number of RAM accesses in longest thread * RCR * 2 clocks)

NOTE

The formula adds 1 RAM accesses for the parameter preload that occurs during TST. There are actually 3 accesses during TST, but only the first one can receive wait-states.

Table 12-1. Longest Threads and RAM Accesses for old TPU Functions

Function	Longest Thread	RAM Accesses
DIO	10	4
ITC	40 (no linking) 42 (linking)	7
OC	40	7
PWM	24	4
SPWM Mode 0 Mode1 Mode 2	 14 18 20 (no linking) 22 (linking)	 4 4 4 4
PMA	94	8
PMM	94	8
PSP Angle-Angle Mode Angle-Time Mode	 76 50	 6 3

Table 12-1. Longest Threads and RAM Accesses for old TPU Functions (continued)

Function	Longest Thread	RAM Accesses
SM[1]	160	21
PPWA		
Mode 0	44	9
Mode 1	50[2]	10
Mode 2	44	9
Mode 3	50	10

1. Assumes one master and one slave. For each additional slave
   a) Add 32 clocks and 2 RAM accesses, and
   b) Add (STEP_RATE_CNT ∗ two clocks)
2. With one channel linked. Add two clocks for each additional channel linked.

12.5.4.3 Mapping the Channels for Each Time Slot

To determine when a channel will be serviced again, it is necessary to determine which other channels will be serviced first. Do this by assuming all channels are continuously requesting service and mapping the channels into the time-slot sequence.

12.5.4.4 Adding Time for Time-Slot Transitions

Add six system clocks for time-slot transitions which occur after each time slot.

12.5.4.5 First-Pass Analysis Worst-Case Latency Examples

The examples in this section assume the system configuration shown in Table 12-2.

Table 12-2. System Configuration Example

Channel	Priority	Function[1, 2]
0	High	PWM (driving a DC motor)
1	Middle	PPWA (Mode 0, measuring the DC motor speed)
2	Low	DIO (Input)

1. 9% RAM Collision Rate (RCR)
2. CPU clock rate = 40 MHz, or 25 ns per clock period

12.5.4.6 Finding the WCL for PWM on Channel 0

The following shows how to find the WCL for PWM on channel 0.

1. Find the worst-case service time for each active channel.
   a) Longest thread of PWM is 24 CPU clocks with four RAM accesses.

24 + ((4 RAM accesses+1) * 0.09 * 2 CPU clock waits) = 24.9 CPU clocks, rounded up to 25 CPU clocks (since there are no partial clock periods)

Channel 0 worst-case service time = 25 CPU clocks.

b) Longest thread of PPWA in mode 0 is 44 CPU clocks with nine RAM accesses.

44 + ((9 RAM accesses+1) * 0.09 * 2 CPU clock waits) = 45.8 CPU clocks, rounded up to 46 CPU clocks

Channel 1 worst-case service time = 46 CPU clocks.

c) Longest thread of DIO is ten CPU clocks with four RAM accesses.

10 + ((4 RAM accesses +1) * 0.09 * 2 CPU clock waits) = 10.9 CPU clocks, rounded up to 11 CPU clocks

Channel 2 worst-case service time = 11 CPU clocks.

2. Assume channel 0 has just been serviced and that channels 1 and 2 are continuously requesting service. Using the H-M-H-L-H-M-H time-slot sequence, map the channels that are granted for each time slot. See Figure 12-6.

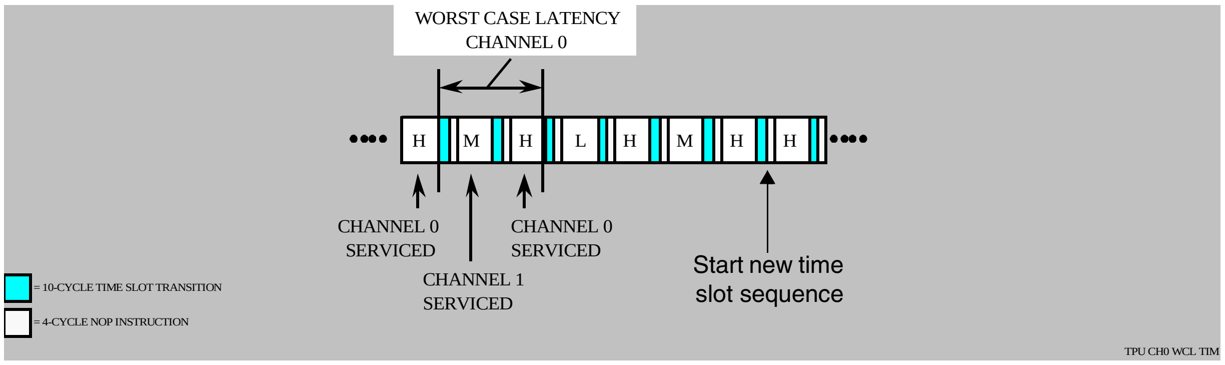


Figure 12-6. Next Servicing for Channel 0

Channel 1 will be serviced in the middle-priority time slot before channel 0 is serviced again.

3. Add time for the six-clock CPU time-slot transitions. See Figure 12-6 and Table 12-3.

A four-clock NOP occurs after each channel is serviced since there is one channel in each priority level, i.e., a new cycle for a priority level is started after each channel is serviced. Time-slot transitions occur after each time slot.

Table 12-3. Worst-Case Latency for Channel 0

Channel 0 worst-case service time	25 clocks
Channel 1 worst-case service time	46 clocks
Two 6-clock time-slot transitions	12 clocks

Table 12-3. Worst-Case Latency for Channel 0

Total clocks	83 clocks

Note: 83 clocks * 25 ns/clock = 2075 ns

Conclusion: in this system configuration PWM can run with a minimum high time or low time of 2075 ns.

Note that in double match eTPU system the PWM can be serviced once in each period, and there is no latency for minimum high time. The latency in eTPU PWM function will represent the minimum PWM period.

12.5.4.7 Finding the WCL for PPWA on Channel 1

The following shows how to find the WCL for PPWA on channel 1.

1. Find the worst-case service time for each active channel. See step 1 of previous example.
2. Assume channel 1 has just been serviced and that channels 0 and 2 are continuously requesting service. Using the H-M-H-L-H-M-H time-slot sequence, map the channels that are granted for each time slot. See Figure 12-7

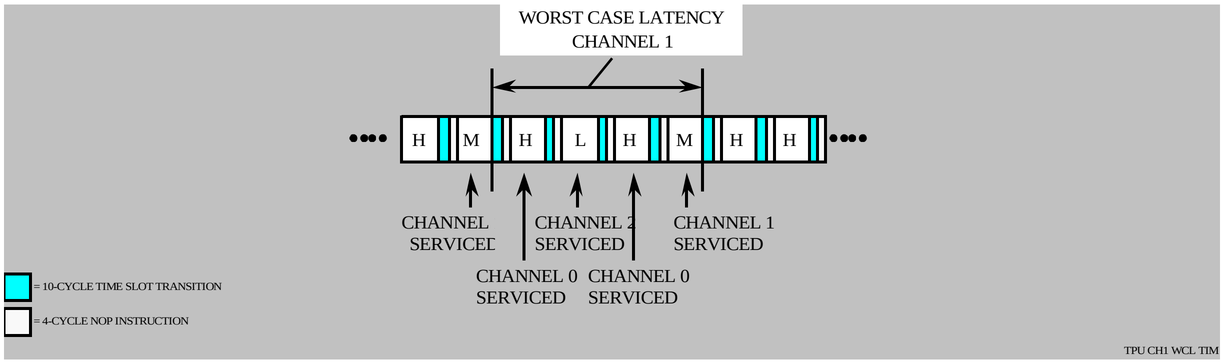


Figure 12-7. Next Servicing for Channel 1

Channel 0 will be serviced twice and channel 2 once before channel 1 is serviced again.

3. Add time for the six-clock CPU time-slot transitions. See Figure 12-7 and Table 12-4.

Table 12-4. Worst Case Latency for Channel 1

Two Channel 0 worst-case service times	50 clocks
Channel 1 worst-case service time	46 clocks
Channel 2 worst-case service time	11 clocks
Four 6-clock time-slot transitions	24 clocks

Table 12-4. Worst Case Latency for Channel 1

Total clocks	131 clocks

Note: 131 clocks * 25 ns/clock = 3275 ns

Conclusion: in this system configuration PPWA can measure a period or pulse of minimum 3275 ns.

Note that PPWA function optimized for eTPU hardware can use double transition mode to measure very narrow pulses with one service after the second transition, and latency will affect only the minimum gap between two input pulses. Also the function threads would have more efficient coding.

12.5.4.8 Finding the WCL for DIO on Channel 2

The following shows how to find the WCL for DIO on channel 2.

1. Find the worst-case service time for each active channel. See step 1 of previous examples.
2. Assume channel 2 has just been serviced and that channels 0 and 1 are continuously requesting service. Using the H-M-H-L-H-M-H time-slot sequence, map the channels that are granted for each time slot. See Figure 12-8.

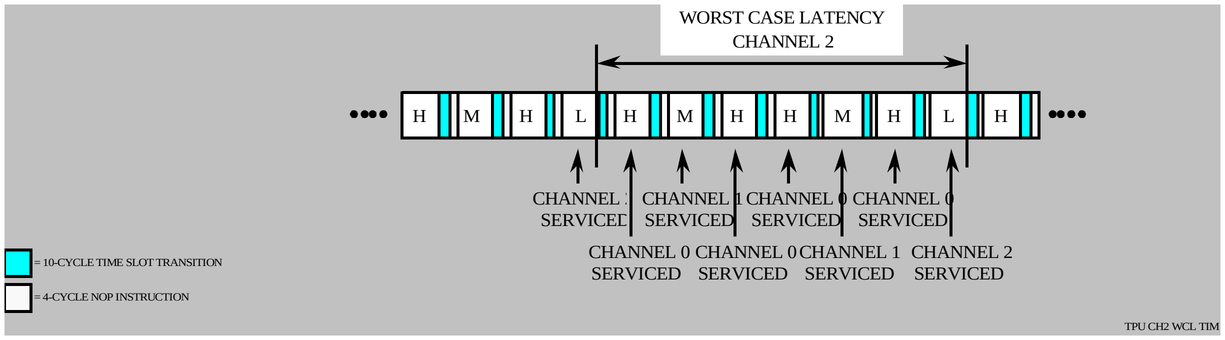


Figure 12-8. Next Servicing for Channel 2

Channel 0 will be serviced four times and channel 1 twice before channel 2 is serviced again.

3. Add time for the 10-clock CPU time-slot transitions. See Figure 12-8 and Table 12-5.

Table 12-5. Worst Case Latency for Channel 2

Four Channel 0 worst-case service times	100 clocks
Two Channel 1 worst-case service time	92 clocks
Channel 2 worst-case service time	11 clocks
Seven 6-clock time-slot transitions	42 clocks

Table 12-5. Worst Case Latency for Channel 2

Total clocks	245 clocks

Note: 245 clocks * 25 ns/clock = 6125 ns

Conclusion: in this system configuration DIO can keep track of the input level at a minimum of every 6125 ns.

Note that DIO function optimized for eTPU hardware can use double transition mode to measure two pin transitions at a time and reduce the service time, improving the overall system performance and latency.

12.5.5 Second-Pass Worst-Case Latency Analysis

The section gives an example of a second-pass analysis for calculating worst-case latency for a channel. The second-pass analysis is useful for higher-performance systems, since it gives a more realistic worst-case latency result than first-pass analysis.

This example uses a relatively simple system in order to illustrate the basic principles of second-pass analysis. For a more complex example of second-pass analysis, refer to, "Multiphase Motor Commutation TPU Function (COMM) (order number: TPUPN09/ D)."

12.5.5.1 Second-Pass Analysis Guidelines

Rather than use a fixed formula, a second-pass analysis relies on the application of the following guidelines.

1. The first-pass analysis makes the assumption that all channels in the system are continually requesting service. For many systems this is an unrealistic assumption. For example, if TCR1 is counting at a rate of 2 MHz (500 ns per count) and a channel is running the DIO function with a match rate of 20,000 TCR1 counts, the DIO will request service every 10 ms (20,000 * 500 ns = 10,000,000 ns or 10 ms). It is therefore unrealistic to assume that the channel running this DIO function is continuously requesting service. Figure out a realistic service request rate for each channel. Time slots can then be mapped to each channel at the real rate of request.
2. If a function is active during system initialization but not during the high-speed running mode of the system, then that system does not need to be included in the high-speed worst-case latency calculations.
3. Use a realistic SPRAM collision rate.
4. Be careful when assigning functions priority levels and channel numbers. Decide which function or functions will be most difficult to perform at the desired level. Assign those channels high priority and low channel numbers. Try different priority and channel assignments to see how it affects the system.

5. The seven-slot sequence of || H | M | H | L | H | M | H || is asymmetrical when put back-to-back with other seven-slot sequences. Note that in the following sequence there are two high-priority slots next to each other:

   || H | M | H | L | H | M | H |||| H | M | H | L | H | M | H ||

6. When mapping out channels to the sequence, choose a worst-case slot to start the mapping. For example, when estimating WCL for a high-priority channel, do not start the mapping in the last high-priority slot in a seven-slot sequence, as that is a best case for a high-priority channel since another high-priority time slot is next.
7. Instead of always using the longest thread in the function as the worst-case state, evaluate the threads in the function that will be used in the system and use the appropriate worst-case threads. For example, in the preceding example of first-pass analysis, the PWM was shown to be able to achieve a high time and low time of 2475 ns under worst-case conditions. This was derived using the longest PWM state of 24 CPU clocks. This longest state is actually thread 2, the thread that is entered after the pin has just gone high. Thread 3, the thread that is entered after the pin has just gone low, requires only 2 CPU clocks. Therefore, in the first-pass example, the high time was correctly derived, but the low time is actually shorter than was estimated.

12.5.5.2 Second-Pass Analysis Example

This example requires three 50% PWM waveforms: one 5 kHz (200 ms/period) and two 50 kHz (20 ms/period), each running DC motors. (Remember that the PWM function requests service from the eTPU after each high time and after each low time, so the eTPU must handle a request every 100 ms for the 5 kHz PWM and every 10 ms for the 50 MHz PWM.)

NOTE

This example uses square waves for simplicity. Note that to use a PWM waveform in the typical way, in which the pulse is modulated, the pulse must not be modulated in a way that violates the worst-case latency requirements.

This example also uses one DIO channel monitoring a signal level every millisecond and one PPWA channel in mode 0 monitoring the speed of the 5-kHz DC motor. The PPWA must measure periods of 5 kHz (200 ms/period).

The CPU is interrupted by the channel running the PPWA function after measuring 200 periods (every 40 ms). The interrupt service routine performs an averaging of the period accumulation and checks it against a known parameter. The interrupt service time is so short and infrequent that it is a tiny fraction of total system time. The interrupt service routine contains no polling of the parameter RAM. Therefore, RCR = 0% is a realistic assumption.

12.5.5.3 First-Try System Configuration

Try a system configuration that seems likely to work. If it does not, change priority levels or channel numbers.

The 5 kHz and 50 kHz PWMs are the most time-critical functions. Those are as-signed high priority. PPWA is assigned middle priority. The DIO is low performance and is assigned low priority. Refer to Table 12-6.

Table 12-6. First-Try System Configuration

Channel	Priority	Function[1, 2]
0	High	PWM at 50 kHz (needs a 4-µs WCL)
1	High	PWM at 50 kHz (needs a 4-µs WCL)
2	High	PWM at 5 kHz (needs a 40-µs WCL)
8	Middle	PPWA at 5 kHz (needs a 80-µs WCL)
15	Low	DIO as input at rate of 1 ms

1. 0% RAM collision rate
2. CPU clock rate = 40 MHz, or 60 ns per clock period

With this system configuration, worst-case service time for each active channel is determined as follows:

a) Longest thread of the PWM is 24 CPU clocks with four RAM accesses.
 24 + (4 RAM accesses+1) * 0 * 2 CPU clock waits) = 24 CPU clocks
 Channels 0–2 worst-case service time = 24 CPU clocks.

b) Longest thread of PPWA in mode 0 is 44 CPU clocks with nine RAM accesses.
 44 + ((9 RAM accesses +1)* 0 * 2 CPU clock waits) = 44 CPU clocks
 Channel 8 worst-case service time = 44 CPU clocks.

c) Longest thread of DIO is 10 CPU clocks with 4 RAM accesses.
 10 + ((4 RAM accesses +1)* 0 * 2 CPU clock waits) = 10 CPU clocks
 Channel 15 worst-case service time = 10 CPU clocks.
 To find the WCL for channel 0, assume channel 0 has just finished service.
 Map the channels in the H-M-H-L-H-M-H sequence. See Figure 12-9.

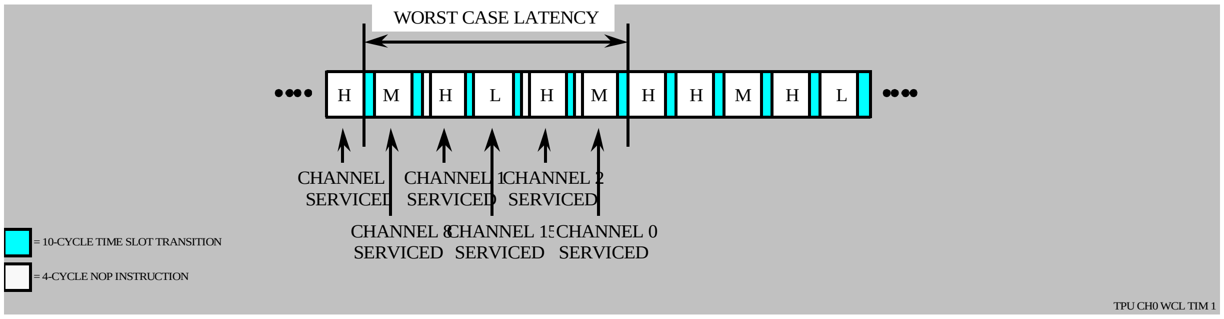


Figure 12-9. Worst-Case Latency for Channel 0 (First Try)

Conclusion: with this system configuration, worst-case latencies for channels 0 and 1 are too high (WCL for channel 1 is the same as WCL for channel 0). Try a different system configuration.

12.5.5.4 Second-Try System Configuration

The second-try system configuration is shown in Table 12-7.

Table 12-7. Second-Try System Configuration

Channel	Priority	Function[1, 2]
0	High	PWM at 50 kHz (needs a 4-µs WCL)
1	High	PWM at 50 kHz (needs a 4-µs WCL)
2	Middle	PWM at 5 kHz (needs a 40-µs WCL)
8	Middle	PPWA at 5 kHz (needs a 80-µs WCL)
15	Low	DIO as input at rate of 1 ms

1. 0% RAM collision rate
2. CPU clock rate = 40 MHz, or 60 ns per clock period

To find the WCL for channel 0, assume channel 0 has just finished service. Map the channels in the H-M-H-L-H-M-H sequence. See Figure 12-10.

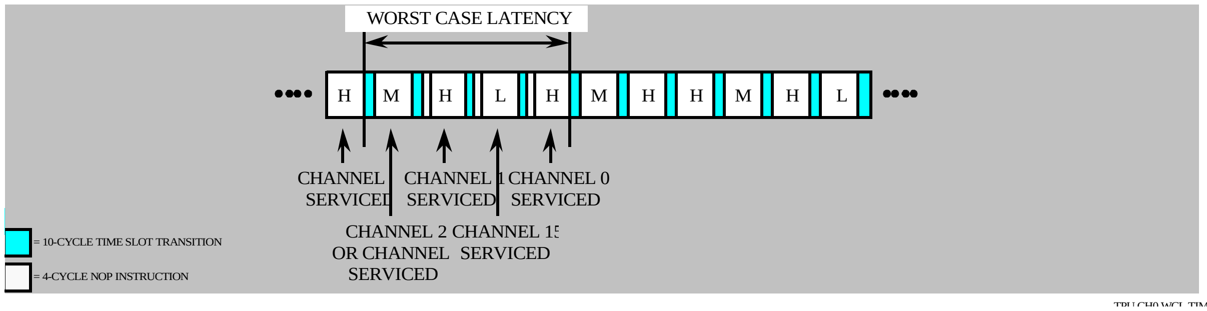


Figure 12-10. Worst-Case Latency for Channel 0 (Second Try)

Conclusion: with this system configuration, the WCL of both channel 0 and channel 1 is 3.85 ms, which is within the limit of 4 ms needed for a 50-kHz PWM.

Next, find the WCL for channel 2. Assume channel 2 has just finished service. Map the channels in the H-M-H-L-H-M-H sequence. See Figure 12-11.

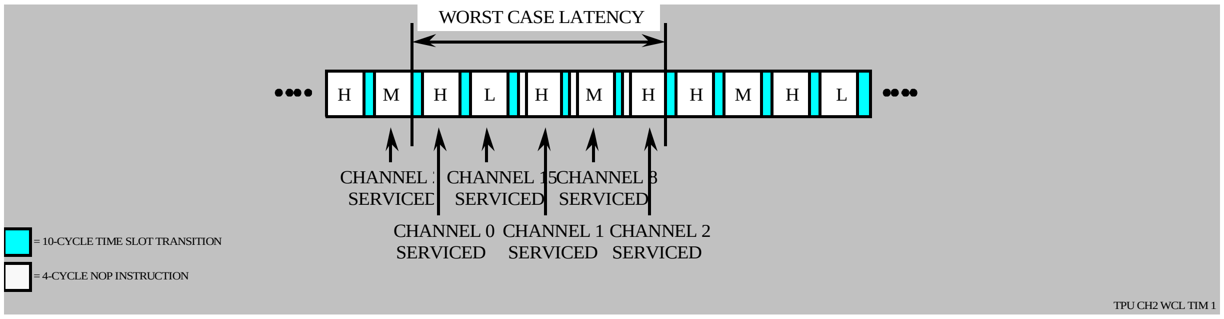


Figure 12-11. Worst-Case Latency for Channel 2

Conclusion: with this system configuration, the WCL for channels 2 and 8 is 4.7 ms, which is within the 40 and 80 ms WCL requirements.

Notice that channels 2 and 8 are well within their WCL requirements. The system could be reconfigured as shown in Table 12-8 to give channels 0 and 1 a larger margin while still keeping channels 2, 8 and 15 within their WCL requirements.

Table 12-8. Second-Try System with Channel 0 and 1 Reconfigured

Channel	Priority	Function[1, 2]
0	High	PWM at 50 kHz (needs a 10-µs WCL)
1	High	PWM at 50 kHz (needs a 10-µs WCL)
2	Middle	PWM at 5 kHz (needs a 40-µs WCL)
8	Low	PPWA at 5 kHz (needs a 80-µs WCL)
15	Low	DIO as input at rate of 1 ms

1. 0% RAM collision rate
2. CPU clock rate = 40 MHz, or 60 ns per clock period

12.6 Endianness

The address of the 24-bit parameters and the most significant byte depends on the endianness of the MCU. Table 12-9 shows the parameter addresses for big and little endian machines.

Table 12-9. Parameter Addresses and Endianness

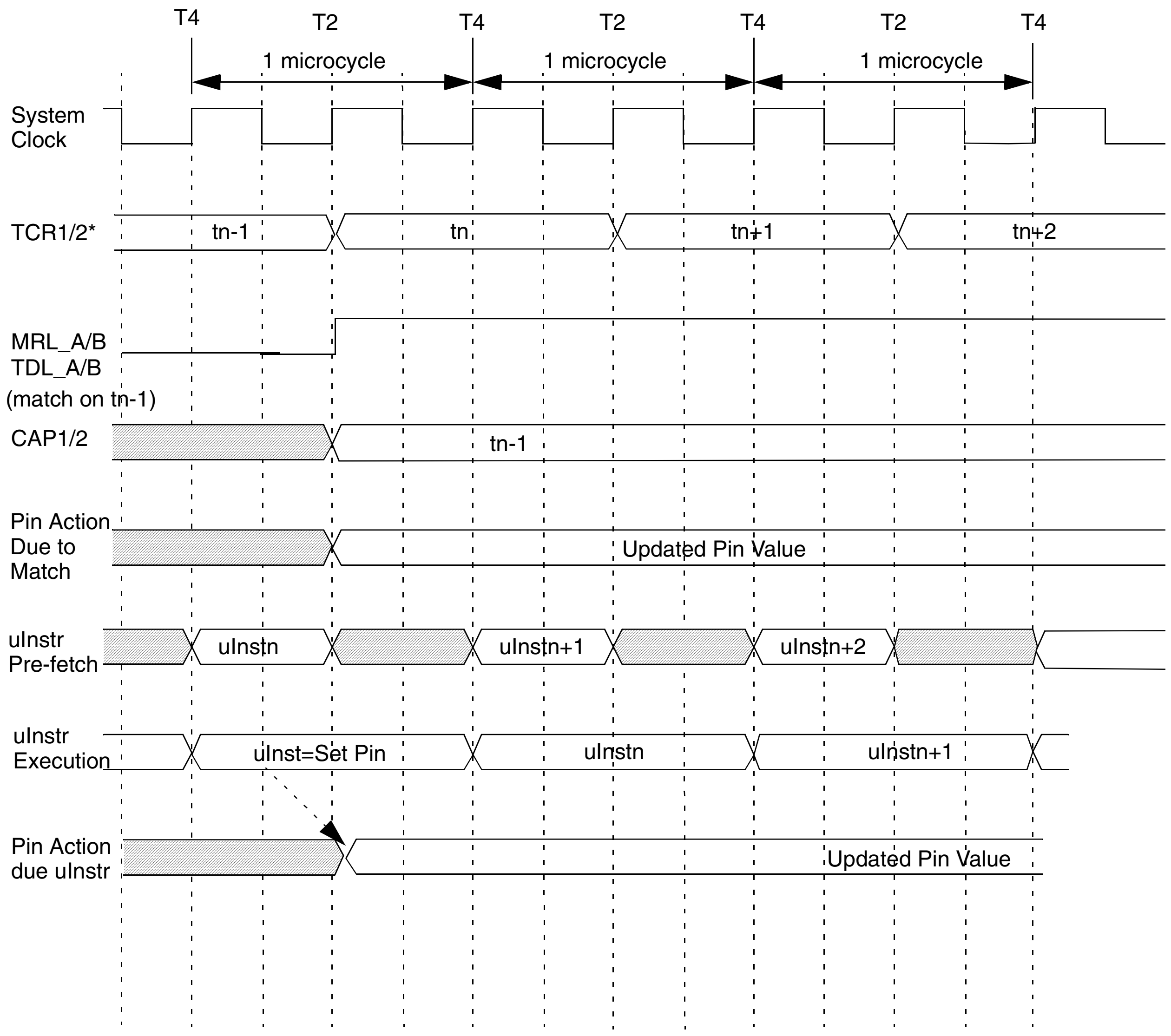
Parameter	Byte Address Offset (n = Word Address Offset)	
	Big Endian	Little Endian
32-bit	4*n	
24-bit	4*n + 1	4*n

Table 12-9. Parameter Addresses and Endianness

Parameter	Byte Address Offset (n = Word Address Offset)	
	Big Endian	Little Endian
32-bit parameter is most significant byte	4*n	4*n + 3
24-bit parameter is most significant byte	4*n + 1	4*n + 2
least significant byte	4*n + 3	4*n

Appendix A
Microcycle Timing

Figure A-1 shows the main timings relevant to the eTPU user.



Note: *TCR clock/prescaler selection = 2 x system clock

Figure A-1. Channel I/O Timing

The sequential occurrence of the four T states (T1 – T4) constitutes a microcycle. Only T2 and T4 are taken as reference for timings, either internal or external. T2 and T4 have the timing of the positive system clock pulses, and are used in most of the eTPU logic in an edge-triggered design style. Some special logics with special timing requirements (like the SPRAM, SCM and synchronizers) also use T1 and T3 to achieve fine timing adjustments.

The eTPU system requires 50% duty cycle on the system clock to meet the RAM discharge and precharge requirements.

Two additional T states are derived from the system clocks: $\overline{\text{T2}}$ and $\overline{\text{T4}}$. $\overline{\text{T2}}$ occurs when the eTPU loses SPRAM arbitration to a bus master or due to attempt to lock a semaphore which is locked by the other eTPU. $\overline{\text{T4}}$ occurs in halt state (due to a breakpoint or ipg_debug assertion, for instance).

$\overline{\text{T2}}$ and $\overline{\text{T4}}$ states are defined as wait states of one system clock in which the T clocks continue to run, but the control signals associated with the clocks are unaffected. That is, no operation occurs during these states. Both $\overline{\text{T2}}$ and $\overline{\text{T4}}$ states occur in multiples of two system clocks to keep the microengine synchronized with the free running channels.

Thus, the eTPU has two types of wait states:

- Wait-T2: Wait for SPRAM access or for SPRAM semaphore lock, from clock pulse T2 until one of the next T2 clock pulses of another microcycle.
- Wait-T4: Wait in debug mode, from clock pulse T4 until one of the next T4 clock pulses of another microcycle.

Figure A-2 and Figure A-3 shows the timing of T2 and T3 wait-states, respectively.

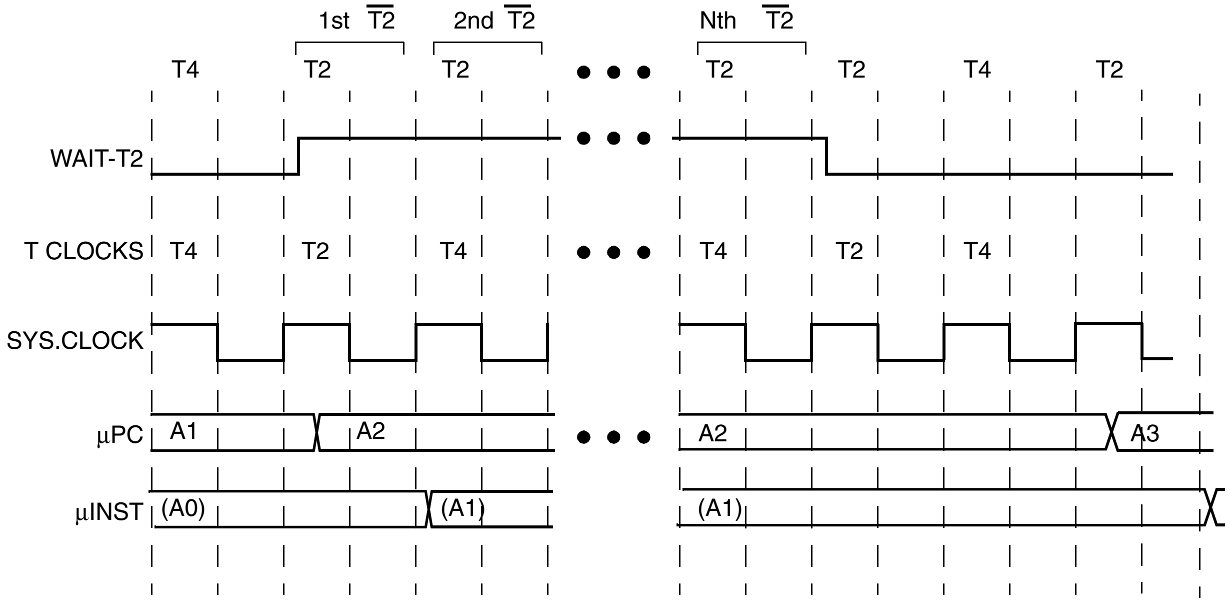


Figure A-2. T2 Wait-State Timing

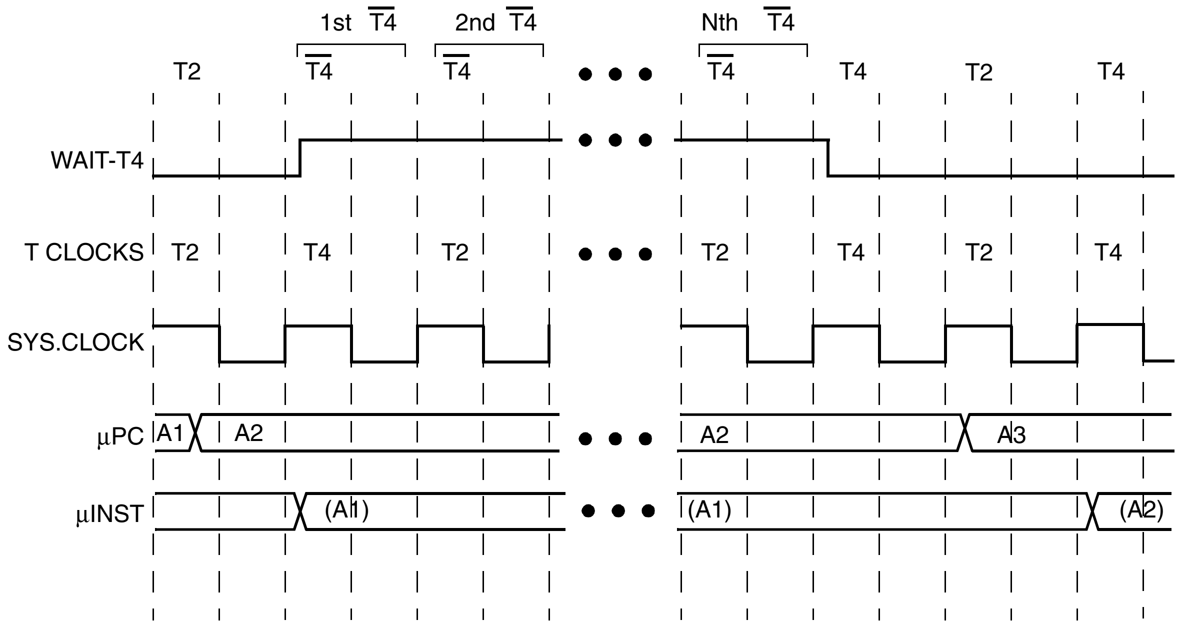


Figure A-3. T4 Wait-State Timing

Appendix B
Initialization Code Example

The code example below initializes eTPU_A engine and configures an eTPU UART function to perform the receiver at channel 1 and the transmitter at channel 0. The function works without parity and the data word is 8 bits in size. The initialization code assumes the microcode function is previously loaded into the SCM.

```
****************************************************************************

// Initilization program for eTPU engine A,

// function microcode previously loaded into SCM.

// No angle mode, eTPU UART function configured to perform at channels 0 and 1.

// Channel0 - Tx_UART

// Channel1 - Rx_UART


// UART Specifications:

// Data word size: 8 bits

// Parity: disabled


// ******************** Definitions ******************************


//Bases

#define ETPU_BASE      0x000           //MCU-dependent

#define SPRAM_BASE     0x000                  //MCU-dependent


//General Configuration Registers

#define ETPUMCR_OFFSET        0x000     //Module configuration register

#define ETPUTBCR_A_OFFSET     0x020     //Time base configuration register
```

```
#define ETPUECR_A_OFFSET      0x014     //Engine configuration register
#define ETPUCIER_A_OFFSET     0x240     //Channel interrupt enable register
#define ETPUCDTRER_A_OFFSET   0x250    //Data transfer interrupt enable register

//channel0 configuration registers
#define ETPUC0CR_A_OFFSET     0x400     //Channel0 configuration register
#define ETPUC0SCR_A_OFFSET    0x404     //Channel0 status control register
#define ETPUC0HSRR_A_OFFSET   0x408    //Channel0 host service request. register

//channel1 configuration registers
#define ETPUC1CR_A_OFFSET     0x410     //Channel1 configuration register
#define ETPUC1SCR_A_OFFSET    0x414     //Channel1 status control register
#define ETPUC1HSRR_A_OFFSET   0x418     //Channel1 status control register

// Tx_UART SPRAM parameters
#define MATCH_RATE_TX_OFFSET   0x004    //Channel0 parameter 1
#define DATA_UART_TX_OFFSET    0x008    //Channel0 parameter 2
#define DATA_SIZE_TX_OFFSET    0x00C    //Channel0 parameter 3

// Rx_UART SPRAM parameters
#define MATCH_RATE_RX_OFFSET  0x024     //Channel1 parameter 1
#define DATA_UART_RX_OFFSET   0x028     //Channel1 parameter 2
#define DATA_SIZE_RX_OFFSET   0x02C     //Channel1 parameter 3

//
#define ETPUMCR            (*((int*)(ETPUMCR_OFFSET + ETPU_BASE)))
#define ETPUTBCR_A         (*((int*)(ETPUTBCR_A_OFFSET + ETPU_BASE)))
#define ETPUECR_A          (*((int*)(ETPUECR_A_OFFSET + ETPU_BASE)))
#define ETPUCIER_A         (*((int*)(ETPUCIER_A_OFFSET + ETPU_BASE)))
#define ETPUCDTRER_A       (*((int*)(ETPUCDTRER_A_OFFSET + ETPU_BASE)))
#define ETPUC0CR_A         (*((int*)(ETPUC0CR_A_OFFSET + ETPU_BASE)))
#define ETPUC0SCR_A        (*((int*)(ETPUC0SCR_A_OFFSET + ETPU_BASE)))
```

```
#define ETPUC0HSRR_A       (*((int*)(ETPUC0HSRR_A_OFFSET + ETPU_BASE)))
#define ETPUC1CR_A         (*((int*)(ETPUC1CR_A_OFFSET + ETPU_BASE)))
#define ETPUC1SCR_A        (*((int*)(ETPUC1SCR_A_OFFSET + ETPU_BASE)))
#define ETPUC1HSRR_A       (*((int*)(ETPUC1HSRR_A_OFFSET + ETPU_BASE)))
#define MATCH_RATE_TX      (*((int*)(MATCH_RATE_TX_OFFSET + SPRAM_BASE)))
#define DATA_UART_TX       (*((int*)(DATA_UART_TX_OFFSET + SPRAM_BASE)))
#define DATA_SIZE_TX       (*((int*)(DATA_SIZE_TX_OFFSET + SPRAM_BASE)))
#define MATCH_RATE_RX      (*((int*)(MATCH_RATE_RX_OFFSET + SPRAM_BASE)))
#define DATA_UART_RX       (*((int*)(DATA_UART_RX_OFFSET + SPRAM_BASE)))
#define DATA_SIZE_RX       (*((int*)(DATA_SIZE_RX_OFFSET + SPRAM_BASE)))


// Macros
#define TCR2_PRESCALER(x)              ((x & 0x3F) <<  8)
#define TCR1_PRESCALER(x)              (x & 0xFF)
#define CHANNEL_FUNCTION(x)            ((x & 0x1F) << 16)
#define CHANNEL_PARAM_BASE_ADDR(x)     (x & 0xFF)
#define FUNCTION_MODE(x)               (x & 0x3)
#define MATCH_RATE_TRANS(x)            (x & 0xFFFF)
#define MATCH_RATE_REC(x)              (x & 0xFFFF)
#define DATA_WORD_Tx(x)                (x & 0x3FFF)
#define DATA_SIZE_TRANS(x)             (x & 0xF)
#define DATA_SIZE_REC(x)               (x & 0xF)
#define HOST_SERV_REQ(x)               (x & 0x7)
#define ENTRY_TABLE_BASE(x)            (x & 0x1F)



//ETPUMCR fields - Module Configuration Register
#define PSE             0x00000002      //Parameter sign extension
#define SCMMISEN        0x00000200      //SCM MISC enable
#define VIS             0x00000040      //SCM visibility
#define GTBE            0x00000001      //Global time base enable
```

```
//ETPUTBCR_A fields - Time Base Configuration Register

#define TCRCLK_FILTER_TWOSAMPLE         0x00000000
//TCRCLK filter in two sample mode

#define TCRCLK_FILTER_INTEGRATION       0x00800000
//TCRCLK filter in integration mode

#define TCRCLK_FILTER_DIV2CLOCK         0x00000000
//TCRCLK filter uses system clock divided by 2

#define TCRCLK_FILTER_CHANNELCLOCK      0x00400000
//TCRCLK filter uses channel clock


#define TCR2_RISE                       0x00100000  //TCR2 inc rising edge

#define TCR2_FALL                       0x00200000  //TCR2 inc falling edge

#define TCR2_RISEFALL                   0x00300000  //TCR2 inc rise and fall

#define TCR2_GATEDDIV8                 0x00000000  //TCRCLK gates system clock/8

#define TCR1CLK_SOURCE_DIV2             0x00000000  //TCR1 source system clock/2

#define TRC1CLK_SOURCE_TCRCLK           0x00040000  //TCR1 source is TCRCLK pin

#define CHANNEL_FILTER_TWOSAMPLEMODE    0x00000000  //Filter: two sample mode

#define CHANNEL_FILTER_THREESAMPLEMODE  0x00008000  //Filter: three sample mode

#define CHANNEL_FILTER_CONTMODE         0x0000C000  //Filter: continuous mode


//ETPUECR fields - Engine Configuration Register

#define FILTER_PRESCALER_CLOCK_DIV4     0x00010000  //System clock/4


//ETPUCxCR fields - channel_x configuration register

#define CHANNEL_INT_ENABLE              0x80000000  //Channel interrupt enable

#define CHANNEL_DATA_TRANSF_REQ_ENABLE  0x40000000
//Channel data transfer req. enable

#define CHANNEL_PRIORITY_DISABLE        0x00000000  //Channel disable

#define CHANNEL_PRIORITY_LOW            0x10000000  //Low priority channel

#define CHANNEL_PRIORITY_MIDDLE         0x20000000  //Middle priority channel

#define CHANNEL_PRIORITY_HIGH           0x30000000  //High priority channel


//DATA_UART - SPRAM

#define CLEAR_TDRE     0x007FFFFF   //TDRE must be zero to signal new valid
```

```
                                //data to be transmitted



void init_etpu( ){


volatile int temp;


//Initialize eTPU module configuration register(ETPUMCR)

ETPUMCR = 0x00070000;   //SCMSIZE is 16K(7 2K blocks)


//Initialize eTPU time base configuration register(ETPUTBCR)

ETPUTBCR_A    =    (TCR1CLK_SOURCE_DIV2    |    CHANNEL_FILTER_TWOSAMPLEMODE    |
TCR1_PRESCALER(8));


//Initialize eTPU engine configuration register(ETPUECR)

ETPUECR_A =  (ENTRY_TABLE_BASE(0x1F) | FILTER_PRESCALER_CLOCK_DIV4);


//Write to the channel configuration Registers(ETPUCxCR) to choose the

//function to be performed by the channel and its parameter
//base address. Standard entry table is selected.

ETPUC0CR_A     =     (CHANNEL_INT_ENABLE     |     CHANNEL_FUNCTION(15)     |
CHANNEL_PARAM_BASE_ADDR(0x00));

ETPUC1CR_A     =     (CHANNEL_INT_ENABLE     |     CHANNEL_FUNCTION(15)     |
CHANNEL_PARAM_BASE_ADDR(0x02));


//Write to the channel status control registers(ETPUCxSCR) to choose

//variations within the function flow.

ETPUC0SCR_A = (FUNCTION_MODE(0));  // no parity for transmitter

ETPUC1SCR_A = (FUNCTION_MODE(0));  // no parity for receiver


//Write to spram for parameter initialization of each configured channel


MATCH_RATE_TX = MATCH_RATE_TRANS(0x412); //setup match rate for transmitter

DATA_UART_TX = DATA_WORD_TX(0x000000AA); //load first byte to be transmitted=AA
```

```
DATA_SIZE_TX = DATA_SIZE_TRANS(8);       //8-bit data word for transmitter
MATCH_RATE_RX = MATCH_RATE_REC(0x412);   //setup match rate for receiver
DATA_SIZE_RX = DATA_SIZE_REC(8);         //8-bit data word for receiver

//Write to channel host service request registers(ETPUCxHSRR) to
//initialize active channels(Channel 0 and 1)
ETPUC0HSRR_A = HOST_SERV_REQ(3);
ETPUC1HSRR_A = HOST_SERV_REQ(2);

//write to channel priority field to enable each channel by
//assigning it a high, middle, or low priority
ETPUC0CR_A =(ETPUC0CR_A | CHANNEL_PRIORITY_HIGH);
ETPUC1CR_A =(ETPUC1CR_A | CHANNEL_PRIORITY_HIGH);

//Monitor channel host service request register for completion
//of initialization
//HSR should be zero in the end of initialization
do
{
     temp = ETPUC0HSRR_A;
} while (temp != 0);

do
{
     temp = ETPUC1HSRR_A;
} while (temp != 0);



//Write GTBE bit to start TCR1 and TCR2 time bases counting
//at the same time
ETPUMCR = (ETPUMCR | GTBE);
```

```
}// end of etpu_initialization routine


************************************************************************
```

Appendix C
Channel Mode Summary

Table C-1 explains channel double match submode functionality by showing all event sequence possibilities. The initial state considered for all submodes is channel flags MRL_A, MRL_B, TDL_A and TDL_B reset. From the initial state one can follow the table and verify how each submode behaves in a determined sequence of events. Note that the actions performed by an event type depend on all previous events following the initial state, for a given channel submode.

There are three columns for each event: one for event type, one for enable/disable actions and one for capture. The event type column can be match1, match2, trans1 and trans2 (for double transition modes). The enable/disable actions column, identified as "[blocks](enables)" in column head, specifies which other events are enabled or disabled. Initially disabled events, specified in "initially blocked" column, are usually enabled by other events.

In double transition submodes, the first transition detected is always considered trans1 and the second is considered trans2. This means that trans1 event actually enables the detection of trans2 event. This is not explicit in the table, since it is a general behavior for all double transition submodes.

A sequence of four events (two matches and two transitions) are necessary to describe the behavior of some channel submodes. When a determined sequence of events has less than four events, the other event columns are left blank.

Cells in an "event type" column that have light-grayed background indicate that a service request is generated. More than one event in the same event sequence can issue service request.

NOTE

The table does not exhaust all possibilities of channel logic event sequences, because it doesn't account for possible microcode interventions. For instance, if matches are blocked by first transition and microcode resets TDL_A, the matches become enabled again, and from this point on the channel behaves as if the first transition had never occurred.

Figure C-1. Channel Mode Summary

Mode	Initially Blocked	1st event			2nd event			3rd event			4th event		
		event type	[blocks] (enables[1])	Capt.	event type	[blocks] (enables)	Capt.	event type	[blocks] (enables)	Capt.	event type	[blocks] (enables)	Capt.
em_nb_st	none	match 1/2	none	1/2	match 2/1	none	2/1	trans 1	[matches]	both	trans 2	none	2
					trans 1	[matches]	both	trans 2	none	2			
		trans 1	[matches]	both	trans 2	none	2						
em_nb_dt	none	match 1/2	none	1/2	match 2/1	none	2/1	trans 1	none	1	trans 2	none	2
		match 1	none	1	trans 1	none	1	match 2	none	2	trans 2	none	2
								trans 2	[match 2]	2			
		match 2	none	2	trans 1	[match 1]	1	trans 2	[match2]	2			
		trans 1	[match 1]	1	match 2	none	2	trans 2	none	2			
					trans 2	[match 2]	2						
em_b_st	none	match 1	[match 2]	both	trans 1	[matches]	both	trans 2	none	2			
		match 2	[match 1]	both									
		trans 1	[matches]	both	trans 2	none	2						
em_b_dt	none	match 1	[match 2]	both	trans 1	none	1	trans 2	none	2			
		match 2	[match 1]	both									
		trans 1	[match 1]	1	match 2	none	2	trans 2	none	2			
					trans 2	[match 2]	2						

Figure C-1. Channel Mode Summary

Mode	Initially Blocked	1st event			2nd event			3rd event			4th event		
		event type	[blocks] (enables[1])	Capt.	event type	[blocks] (enables)	Capt.	event type	[blocks] (enables)	Capt.	event type	[blocks] (enables)	Capt.
bm_st	none	match 1/2	none	1/2	match 2/1	none	2/1	trans 1	[matches]	both	trans 2	none	2
					trans 1	[matches]	both	trans 2	none	2			
		trans 1	[matches]	both	trans 2	none	2						
bm_dt	none	match 1/2	none	1/2	match 2/1	none	2/1	trans 1	none	1	trans 2	[matches]	2
					trans 1	none	1	match 2/1	none	2/ none	trans 2	[matches]	2
								trans 2	[matches]	2			
		trans 1	none	1	match 1/2	none	none/2	match 2/1	none	2/ none	trans 2	[matches]	2
								trans 2	[matches]	2			
					trans 2	[matches]	2						
m2_st	trans 1	match 1	(trans 1)	1	match 2	none	2	trans 1	[matches]	both	trans 2	none	2
					trans 1	[matches]	both	trans 2	none	2			
		match 1 and match 2	(trans 1)	both	trans 1	[matches]	both	trans 2	none	2			
		match 2	[match 1]	2									
m2_dt	trans 1	match 1	(trans 1)	1	trans 1	none	1	trans 2	[match 2]	2			
								match 2	none	2	trans 2	none	2
					match 2	none	2	trans 1	none	1	trans 2	none	2
		match 1 and match2	(trans 1)	both	trans 1	none	1	trans 2	none	2			
		match 2	[match 1]	2									

Figure C-1. Channel Mode Summary

Mode	Initially Blocked	1st event			2nd event			3rd event			4th event		
		event type	[blocks] (enables[1])	Capt.	event type	[blocks] (enables)	Capt.	event type	[blocks] (enables)	Capt.	event type	[blocks] (enables)	Capt.
m2_o_st	trans 1 match 2	match 1	(match 2) (trans 1)	1	▨ trans 1	[matches]	both	trans 2	none	2			
					▨ match 2	[trans 1]	2						
m2_o_dt	trans 1 match 2	match 1	(match 2) (trans 1)	1	trans 1	none	1	▨ trans 2	[match 2]	2			
								▨ match 2	[trans 2]	2			
					▨ match 2	[trans 1]	2						
sm_st[2]	match 2	▨ match 1	none	both	▨ trans 1	none	both	trans 2	none	2			
		▨ trans 1	[match 1]	both	trans 2	none	2						
sm_dt	match 2	▨ match 1	none	both	trans 1	none	1	▨ trans 2	none	2			
		trans 1	none	1	▨ match 1	none	2	▨ trans 2	none	2			
		trans 1	none	1	▨ trans 2	[match1]	2						
sm_st_e[3]	match 2	▨ match 1	none	1	▨ trans 1	none	1						
	trans 2	▨ trans 1	[match 1]	1									

▨ Generates Service Request

1. Transition 1 always enables Transition 2
2. sm_st is compatible with TPU3 channel logic.
3. It is not possible to include all functionality of this submode in table. See Section 5.5.4.9, “Single Match Enhanced Mode (sm_st_e),” for more details.

Appendix D
eTPU MISC Algorithm

The MISC generator is based on the following polynomial:

$G(x) = 1 + x^1 + x^2 + x^{22} + x^{31}$ = 0x80400007

The MISC signature generation starts by clearing the MISC accumulator value to 0 and preloading the MISC counter with the highest SCM address. It then steps through each address decrementing the counter, reading 32 bit values and following the algorithm below:

```
If the least significant bit in MISC is 1 then
     MISC = MISC right shifted by 1 bit
     MISC = MISC XOR 0x80400007
else
     MISC = MISC right shifted by 1 bit
end if
MISC = MISC XOR RAM data
```

The code example below shows an excerpt of C code that calculates the MISC signature for a given array of data, based on the previous algorithm:

```
#define SCM_size   (MAX_SCM_ADDRESS)
#define POLY       0x80400007            /* G(x) = 1 + x1 + x2 + x22 + x31 */

/*****************************************************************
 FUNCTION : void calc_misc()
 PURPOSE : This function calculates the MISC value.
 INPUTS NOTES : none
 RETURNS NOTES : MISC value
 GENERAL NOTES : the array 'unsigned int data[]' represents the actual memory
                 array, organized in 32-bit words.
```

```
*********************************************************************/
int calc_misc (void)
{
int j; /* loop counter */

unsigned int misc = 0;
misc = SCM_size-1; /* according to the algorithm, shouldn't this go here? */
for (j = (SCM_size-1); j >= 0 ; j-=4) {   /* SCM_size has the number of in SCM */

      if (misc & 0x1) {
          misc >>= 1;
          misc ^= POLY;
      }
      else {
      misc >>= 1;
      }
      misc ^= data[j];           /* data[j] is the actual 32-bit word (byte
      addressed) taken from the SCM array */
}

return (misc);          /* final signature calculated */

};
```

The value calculated by this algorithm must be loaded into register ETPUMISCCMPR prior to activating the SCM MISC calculator in eTPU. Once the MISC calculator is activated (bit SCMMISEN in register ETPUMCR is written to 1) eTPU itself will start this procedure, see Note: , reading the SCM whenever allowed by microengine. At the end of the cycle, when all of the array has been read and the SCM signature is calculated, the host CPU can be notified via Global Exception if the MISC accumulator does not match the value in ETPUMISCCMPR.

NOTE

eTPU MISC hardware is optimized to read 32-bit words from memory and to calculate this CRC in parallel, rather than shifting one bit at a time. The actual implementation inside eTPU arrives at the same results, though it doesn't exactly match exactly the algorithm shown here.

Further detail on MISC calculation can be found on Section 5.10.3.1, "SCM Test for MISC (Multiple Input Signature Calculator)." The application note, "AN2192 - Detecting Errors in the Dual Port RAM (DPTRAM) Module," is also a good source of MISC signature information (though it refers to the TPU).

The average time taken by MISC to complete the signature of the whole SCM can be given by the formula:

```
Average MISC period = S / (4 * f * (1 - L))
```

where f is clock frequency, S is SCM size in bytes and L is eTPU load (as a percentage of execution clocks over a period of time, including TST clocks).