Freescale Semiconductor
Application Note

AN2933
Rev. 0, 12/2004

Understanding the eTPU Channel Hardware

by: Mike Pauwels
TECD Systems Engineer

This is one of a series of application notes intended to help the microcontroller systems engineer to design and implement code for the Enhanced Time Processor Unit (eTPU).

This note describes the architecture of the eTPU channel hardware, and demonstrates how the action units can be configured to meet system requirements and accurately controlled for flawless operation.

Most of the examples shown below are taken from software that has been designed and tested on the eTPU hardware and represent the state of the art.

Table of Contents

1 Overview

The eTPU is an autonomous slave processor offered on various families of Freescale microcontrollers. It is a significant enhancement of the successful Time Processor Unit (TPU) that has found applications in automotive, electromechanical, communications, and other systems for many years. The extent of the eTPU enhancements have precluded code compatibility with the TPU, but the architecture was designed with a similar

approach, and most TPU applications can be easily ported to the eTPU.

The TPU was programmed in microcode, and the complexity of the language made the device difficult to learn and tedious to program. The enhancements in the eTPU have made the newer device considerably more complex to use than the TPU. To offset the increased complexity, Freescale has teamed with tools vendors to provide a set of eTPU development tools to enable the engineer to program the device in C. This has proved to increase code productivity 2-3 times, while ensuring that future devices can be source code compatible with eTPU.

The three major steps in the development of an eTPU system are the following:

1. Interfacing the channel control logic to the application
2. Writing the eTPU procedural code to control the channel logic
3. Interfacing the eTPU code to the host application

This note addresses the first step of that process. In fact, understanding the channel logic is the key to applying the eTPU in a wide range of systems.

2 Architecture of the eTPU Channel

Like the TPU, the eTPU is organized as a single processing engine tightly coupled to a number of channels each with independent logic interfaced to input and output pins. In the TPU channel, each input/output pin was connected to an output compare and an input capture register. The enhancements in the eTPU include separating the input and output signal, providing two action units per channel each with an input capture and output compare, and providing a number of operating modes that combine the matches and compares for complex logic interface to the pins.

A block diagram of the eTPU Channel Logic is shown in Figure 1 To understand this diagram, it is useful to consider the several subsystems separately.

2.1 Timer/Counter Registers

There are two timer counter registers in the eTPU that are available to all channels. These are the bases for all captures and matches in the action units. There are some differences in the manner in which they are driven. TCR1 can be derived from the system clock through a prescaler or can be driven from an external source through the TCRCLK pin or from an external server through the STAC bus. Typically TCR1 is derived from the system clock.

TCR2 can effectively be driven by the same sources as TCR1, plus it can be driven by special Angle Clock logic in the eTPU. In this mode, the TCR2 tracks the position of a rotating wheel such as might be found on an automotive engine or a motor shaft encoder. In examples below, the TCR2 value is often referred to as the *angle*.

The TCRs are 24 bits wide. In counting mode, they roll over from 0xFFFFFF to 0x000000 and continue counting. In angle mode, the counter can be set to reset to zero periodically, corresponding to the revolution of the wheel. The TCR counters may be written by eTPU software or frozen on execution of a register write. The TCRs can be read but not written by the host.

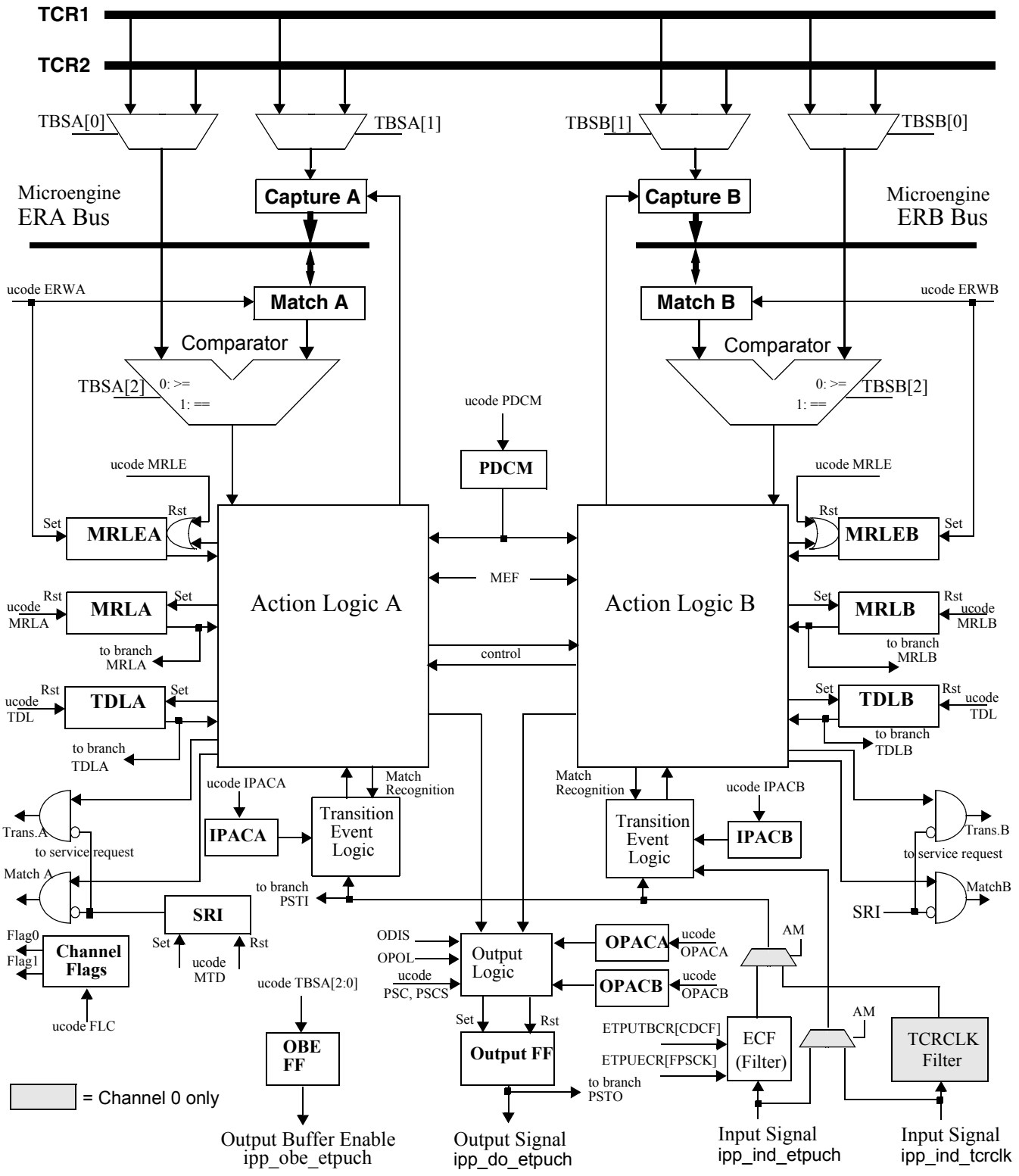

Figure 1. eTPU Channel Logic Block Diagram

2.2 Action Units

There are two sets of match and capture registers associated with each channel. These are referred to as *Action Unit A* and *Action Unit B* and the corresponding match and capture registers are also designated *A* and *B*. Earlier documentation used the numbers 1 and 2 respectively to distinguish these units, and some older software samples and primitive identifiers may continue to use the old designation.

The action units include one each match and capture registers, and associated latches which are described below.

2.2.1 Match Registers

Action units A and B are similar but not identical. Each one has a match register and a capture register, which are identical. The differences in the action units are controlled by the channel mode logic configuration.

The match registers consist of a register holding a match value and a comparators continuously comparing the register value with a selected TCR. The comparators can operate either in an *equals-only* or a *greater-than-or-equal* mode. In *equals-only*, the comparators asserts on a clock cycle when the selected TCR exactly equals the register value. In *greater-than-or-equal*, the comparator makes a relative comparison of the register value with the TCR. The match is satisfied if the TCR is equal to or no more than 0x7FFFFF greater than the register value. For a detailed discussion of this comparison, please see the Reference Manual.

When either Match Register A or Match Register B comparator asserts, it sets the corresponding match recognition latch *MatchALatch* (*MRLA)* or *MatchBLatch* (*MRLB)*.

2.2.2 Capture Registers

Pin transitions or matches can cause the capture registers in the action units to latch the value of either TCR1 or TCR2 at the time of the match event or pin transition. In general, a TCR value can be captured on a match--for example, to catch the time a certain angle triggered action occurred--but if a programmed transition is detected, the capture value will overwrite a previously captured match. Also, a first transition might capture both TCRs, but a second transition may overwrite one of them. The exact details on what is captured on which event is described in Section 3, “Channel Modes.”

2.2.3 Transition Detect Latches

A pin action which captures one or more TCRs will also set the corresponding transition detect latch (TDL), *TransitionALatch* (*TDLA*) or *TransitionBLatch* (*TDLB*).

The TDLs are always set coherently with the capture of the TCR in the capture register, so if the TDL is set, the capture register will hold the value in the TCR at the time of the transition. Note that service may have been requested by a match or previous transition when a transition occurs. When the scheduler grants service to a channel, the eTPU executes a Time Slot Transition (TST), during which it sets up the environment for the channel service. As part of the TST, the eTPU latches the state of the TDLs for

subsequent testing by the software. At the same time during the TST, the eTPU reads the value in the Capture registers into the ERT registers.

A TDL set after the TST latching cannot be tested by the eTPU during the service routine unless the CHAN register is rewritten. Whenever the CHAN is written with a channel number, the values of the TDLs, MRLs, and Capture Registers are updated to the current values in the corresponding channel.

2.2.4 Match Recognition Latches

For each match register, there is a corresponding match recognition latch (MRL) which is set by the assertion of the comparator. The MRL set signal is normally gated by other signals depending on the selected channel mode. Once set, the match recognition latch may be cleared only by an eTPU software command.

At the TST, the individual MRLs are latched into a condition code register and may be tested by the software as branch conditions. Like the TDLs, the MRLs are not automatically updated for matches occurring during the service routine, but must be specifically updated by writing the CHAN register

The MRLs and TDLs are used to form the entry vector of a thread when a channel requests service. As entry conditions, *MatchALatch* is ORed with *TransitionBLatch* and *MatchBLatch* is ORed with *TransitionALatch*. When a software thread is entered using these channel conditions, the software must test the individual latches to determine which state action to execute.

NOTE

The term *thread* is used to denote the sequence of code from an entry vector resulting from a service request through the execution of the *end* statement. The *end* is normally inserted by the compiler at the final brace (}) after an entry block. A thread typically include several functional states of the system design.

Example 1.

```
if (MatchA_TransB)    // Entry condition -- start of thread
 {
   if (MatchA)        // Conditional branch code
    {
      /* Do the Match A state. */
      ClearMatchALatch();
    }
   else if (TransB)    // Condition branch code
    {
      /* Do the Trans B state. */
      ClearTransLatch();  //Clears ALL Transition Latches
    }
  } // end of thread
```

NOTE

One instruction, `ClearTransLatch()`, clears both TDLs. Because transitions are ordered and TDLs cannot be individually cleared, there is no situation where *TransitionALatch* is cleared and *TransitionBLatch* is set.

2.2.5 Match Recognition Latch Enables

Associated with each MRL is a match recognition latch enable (MRLE). This latch is set only by a software write to the corresponding match register. It is cleared by any set of conditions that sets the MRL, possibly ORed with another channel signal, depending on the channel mode.

The MRLE is always a condition for setting the MRL. Therefore, since the MRL always clears the MRLE, all matches are self-blocking, and can occur only once before being attended to by the software.

Example 2.

```
erta = MyMatchValue;     //ert only is used to write Match Registers
EnableMatchA();          // Writes erta to MatchRegisterA and sets MRLEA
ClearMatchALatch();      // Clears a previously set MRL
```

NOTE

If *EnableMatchA()* and *ClearMatchALatch()* are written in consecutive order, the compiler will pack both operations into a single instruction, and the channel will not miss a match on the new condition.

2.2.6 Action Unit Setup

The eTPU software can write to the action unit timebase select registers (TBSA/TBSB) to determine the TCR interfaced to each match and capture register, and to select whether the matches are done on a greater or equal or an equals only basis.

Table 1. Action Unit Time Base Select

TBSA/TBSA	Macro	Action
Mtcr1_Ctcr1_ge	TimeBaseAMatchTcr1CaptureTcr1GreaterEqual() TimeBaseBMatchTcr1CaptureTcr1GreaterEqual()	Capture TCR1 and greater_or_equal compare to TCR1 for this action unit
Mtcr2_Ctcr1_ge	TimeBaseAMatchTcr2CaptureTcr1GreaterEqual() TimeBaseBMatchTcr2CaptureTcr1GreaterEqual()	Capture TCR1 and greater_or_equal compare to TCR2 for this action unit
Mtcr1_Ctcr2_ge	TimeBaseAMatchTcr1CaptureTcr2GreaterEqual() TimeBaseBMatchTcr1CaptureTcr2GreaterEqual()	Capture TCR2 and greater_or_equal compare to TCR1 for this action unit

Table 1. Action Unit Time Base Select (continued)

TBSA/TBSA	Macro	Action
Mtcr2_Ctcr2_ge	TimeBaseAMatchTcr2CaptureTcr2GreaterEqual() TimeBaseBMatchTcr2CaptureTcr2GreaterEqual()	Capture TCR2 and greater_or_equal compare to TCR2 for this action unit
Mtcr1_Ctcr1_eq	TimeBaseAMatchTcr1CaptureTcr1ExactlyEqual() TimeBaseBMatchTcr1CaptureTcr1ExactlyEqual()	Capture TCR1 and exactly_equal compare to TCR1 for this action unit
Mtcr2_Ctcr1_eq	TimeBaseAMatchTcr2CaptureTcr1ExactlyEqual() TimeBaseBMatchTcr2CaptureTcr1ExactlyEqual()	Capture TCR1 and exactly_equal compare to TCR2 for this action unit
Mtcr1_Ctcr2_eq	TimeBaseAMatchTcr1CaptureTcr2ExactlyEqual() TimeBaseBMatchTcr1CaptureTcr2ExactlyEqual()	Capture TCR2 and exactly_equal compare to TCR1 for this action unit
Mtcr2_Ctcr2_eq	TimeBaseAMatchTcr2CaptureTcr2ExactlyEqual() TimeBaseBMatchTcr2CaptureTcr2ExactlyEqual()	Capture TCR2 and exactly_equal compare to TCR2 for this action unit

2.3 Pin Interface

NOTE

The term “pin” usually modified here with “input” or “output,” refers to the interconnection point between the eTPU module and the rest of the MCU silicon. These pins may or may not be connected to the MCU I/O pins. They may be tied together at the MCU pin, or they may be chain connected such that eTPU input for channel X is tied to the eTPU output for channel Y and so on. The examples and information presented herein does not make any assumption about the MCU interconnection. See the product information for eTPU pin interconnection for a specific device.

There are separate controls on the input and output pins of each channel, therefore there is no data direction control in the eTPU instruction set. Input instructions control the input pin and output instructions control the output pin. If a match drives an output pin and that pin is interconnected to an input pin of the same or different channel, the input logic can detect the match drive as a transition.

2.3.1 Output Pin Action

The eTPU channel can drive the output pin by software command or by channel action according to the Output Pin Action Control (OPAC). The most common way to drive the pin is by satisfying a match register comparison in the comparator of one of the action units. The output pin action that occurs as a result of a match is determined by a software write to a register as defined in Table 2.

Table 2. OPACA and OPACB Pin Control

OPACA/OPACB	Macro	Action
match_no_change	OnMatchAPinNoChange() OnMatchBPinNoChange()	No pin change on match
match_high	OnMatchAPinHigh() OnMatchBPinHigh()	Pin switches high on match
match_low	OnMatchAPinLow() OnMatchBPinLow()	Pin switches low on match
match_toggle	OnMatchAPinToggle() OnMatchBPinToggle()	Pin changes state on match
transition_low	OnInputActionAPinLow() OnInputActionBPinLow()	Output pin goes low on input transition
transition_high	OnInputActionAPinHigh() OnInputActionBPinHigh()	Output pin goes high on input transition
transition_toggle	OnInputActionAPinToggle() OnInputActionBPinToggle()	Output pin toggles on input transition

The last three states are useful if the input and output pins are separately connected at the MCU interface. In each of these cases, the output pin transition is directly driven by detection of the programmed input pin transition. This happens in eTPU channel hardware and does not require software intervention.

Note that since the two action units of a given channel can be programmed differently, it is possible for opposite pin actions to be commanded simultaneously. The resulting action is dependent on the channel mode, described in Section 3, “Channel Modes.”

In some input functions depending on the channel mode (see Section 3, “Channel Modes”), the match registers are used to gate or condition the input signal. If the input pin and output pin are tied together, the OPAC for those matches may be set to match_no_change (the default) so that the matches will have no affect on pin state.

The software may also command the output pin directly. When one of the instructions in Table 3 is executed, the output pin of the channel selected by the CHAN register is immediately affected.

Table 3. Immediate Pin State Control

instruction	Macro	Action
PIN = force_pin_high	SetPinHigh()	Pin switches high immediately
PIN = force_pin_low	SetPinLow()	Pin switches low immediately
PIN=set_pin_per_opacA	SetPinPerPacA()	Pin switches according to OPACA
PIN=set_pin_per_opacB	SetPinPerPacB()	Pin switches according to OPACB

2.3.2 Input Pin Action

The channel hardware can be set up to detect transitions on the input pin according to the Input Pin Action Control (IPAC). In general, a pin transition that satisfies the IPAC for an action unit will result in the capture of one of the TCR values in the corresponding capture register. See Section 3, "Channel Modes" for details on specific channel modes. Input actions that are detected by the action unit logic are determined by software writes to a channel register as defined in Table 4

Table 4. IPACA and IPACB Pin Sensing

IPACA/IPACB	Macro	Action
no_detect	DetectADisable() DetectBDisable()	Do not detect transitions
low_high	DetectARisingEdge() DetectBRisingEdge()	Detect a pin rising transition with this action unit
high_low	DetectAFallingEdge() DetectBFallingEdge()	Detect a pin falling transition with this action unit
any_trans	DetectAAnyEdge() DetectBAnyEdge()	Detect any pin transition with this action unit
detect_input_0_on_match	DetectALowOnMatchA() DetectBLowOnMatchB()	Detect if the pin is low when this action unit matches
detect_input_1_on_match	DetectAHighOnMatchA() DetectBHighOnMatchB())	Detect if the pin is high when this action unit matches

The last two selections are useful when a pin state must be sampled at a specific time or angle that can be setup in the match register. When the MLR conditions are satisfied, the *TransitionALatch* (*TDLA*) or *TransitionBLatch* (*TDLB*) will be set if the input pin is at the selected state.

The software may sense the current pin levels--input or output--at any time. The variables *CurrentOutputPin* and *CurrentInputPin* can be tested at any time. The conditions *IsCurrentInputPinHigh()* and *IsCurrentOutputPinHigh()* can be used for conditional branching.

In the TPU, the pin state was sampled at the time slot transition to a new thread or whenever the CHAN register was written by microcode. This latched state, PSS, could be tested at any time in the thread, and was unchanged even if the current pin state changed.

In the eTPU, the current state of either the input pin or the output pin can be tested at any time by the software using *IsCurrentInputPinHigh()* or *IsCurrentOutputPinHigh()*. In addition, the pin state sampled at the TST is made available to the software to maintain compatibility with the TPU. Since, in the eTPU, there is a possibility that the input pin state is different from the output pin state, the selection of which pin is sampled is determined by the host for each channel using the ETPUCxCR bit ETPD. The default selection is the input pin state. The eTPU software can use the macro *IsSampledInputPinHigh()* as a branch condition at anytime. Remember that the pin will be sampled whenever the CHAN register is written, so an assignment *CHAN = CHAN* will re-sample the pin state.

2.3.3 Pin Filters

Every channel pin has an input filter designed to reject fast noise pulses. The filter is programmed identically for all eTPU input pins, except optionally the Angle Clock. When the filter is programmed to reject pulses faster than a couple of clock periods, it will also delay the recognition of an input transition by a similar amount of time. See the Reference Manual for details.

2.3.4 Output Pin Buffers

The eTPU provides software control of pin output buffers for each channel. Each buffer, when enabled, effectively connects the eTPU output to the MCU pin. When the buffer is disabled, the output drive is tri-stated. However, in some implementations, the MCU does not provide connection from the eTPU to the output buffers enable control. In such cases the eTPU output may be controlled by another facility on the MCU. See the MCU Reference Manual for details.

2.4 Other Channel Resources

In addition to the action units and pin logic, each channel has two flags and two function mode bits.

2.4.1 Channel Flags

There are two channel flags associated with each channel. These flags, designated flag0 and flag1, can be set and cleared by the eTPU software, and can be used to qualify the entry vectors. They cannot be read or tested by the eTPU engine.

2.4.2 Function Modes

Not shown in the channel block diagram, there are also two function mode bits associated with each channel. There may be set or cleared by the host only, and tested by the eTPU.

2.5 Channel Service Requests

The channel hardware is one of the three sources of service requests that can be made for an eTPU channel, along with host service requests (HSRs) and links from other channels. Depending on the channel mode and software, transitions and matches can result in channel service requests only after they are enabled by eTPU software. Therefore a channel cannot respond to a hardware request until it has received at least one HSR or link, and executed a channel initialization thread. Somewhere in that thread should be the macro: *EnableMatchAndTransitionEventHandling()* or *EnableEventHandling()*. No channel service requests will be granted by the eTPU scheduler until that instruction is executed. The channel service requests can also be disabled by executing *DisableMatchAndTransitionEventHandling()* or, alternately *DisableEventHandling()*.

2.5.1 Requests to the Scheduler

Service requests from any source are made to the eTPU Scheduler. When the Scheduler grants a service request for a channel, it selects an entry vector according to the conditions that are present when the request is granted. Details of this operation are covered elsewhere, but the following points are important to understand the response of the eTPU engine to a channel service request:

- Regardless of the order of the requests, multiple service requests from a channel are granted according to the conditions that exist at the time of granting.
- The entry vector selected as a result of a service request grant is statically determined by the selected entry vector table (see the Reference Manual). It is clear that if an HSR is asserted at the time of request grant, that will uniquely determine the vector, because where HSR bits are non-zero, channel and link bits are “don’t Care”.
- If there are no HSR bits asserted, a channel request or a link is required to raise a service request. Additional bits such as pin state or flags may be used to further qualify the state.
- The assertion of a match recognition latch in Action Unit A is ORed with the assertion of a transition detect latch in Action Unit B. Likewise, the assertion of a match recognition latch in Action Unit B is ORed with the assertion of a transition detect latch in Action Unit A. In many cases this will be sufficient to resolve the exact thread that needs to be executed, but if not, the software must further resolve the uncertainty.
- All service requests remain until the source is cleared. HSRs are cleared by execution of an end statement in the software, which is usually caused by closing the final brace in the C source for the entry. Link, match and transition flags must be explicitly cleared in the software. If they are not cleared, the scheduler will re-schedule the request after the thread is complete.
- If multiple sources have requested action, the software may execute one thread, clearing one source and exit and leaving the rest of the requests to their own schedule and execute cycle. If time is critical, the software can be designed to test the additional service request sources before the original thread is ended, and execute additional code to service those requests.
- Any thread can clear any or all pending request conditions.

Example 3.

```
...
else if (matchB_transA) // Here on Match 2 (stall) or Transition 1 (Tooth edge)
{
        if (IsTransALatched())   //If the Tooth edge is detected
        {
         /* Here we do the routine to capture and process the input transition. This
lengthy thread from the Automotive Reference Design is not quoted here except to
illustrate the handling of multiple service request sources.
         */
         ClearTransLatch(); // Clears the Transition latches
        SetupMatch_B((EdgeCaptureTime + StallPeriod), Mtcr1_Ctcr1_ge,
match_no_change); //this also clears a MatchB latch
```

```
            }     // end: If the Tooth edge is detected.
            else   // stall--here the transition latch is not set
            {
            DisableMatch();
            CrankStatus = Stall;
            SetChannelInterrupt();//Inform the host that the crank status is updated
            ClearAllLatches();
            }
      }
      ...
```

In the example either a Stall time-out or a Tooth transition or both will cause the thread to be entered. Since the logic first checks the tooth, that branch will be taken when both conditions exist. In that branch, the stall condition is cleared and the stall condition is reset to a later time.

If the stall match only is set, the second branch is taken. The software is now no longer interested in a transition that came too late, and clears all latches before reverting to the stall state.

NOTE

In some channel modes, certain match recognition latches or transition detect latches may be asserted but not request service. However, the latch conditions are nevertheless used to determine the selected vector

2.5.2 Coding the Service Request Handling

Because the Scheduler does not order the service requests for a channel, it is important for the coder to understand the way the vectors are assigned to avoid writing code that handles the requests out of order.

Example 4.

```
/* The following might be used with a Periodic Angle Clock (See Application Note #xxxx)
to produce a short pulse at one angle and a channel interrupt to the host at another
angle. */
/* ... In the setup routine... */
SetupMatchB(Pulse_Start_Angle); //start a pulse at a specified angle
OnMatchBPinHigh();
SetupMatchA(Pulse_Start_Angle + Pulse_Width); //this sets the end of the pulse
OnMatchAPinLow();
Next_Match = Interrupt;
/* ... more... */
if (MatchA_TransB && Flag0)
{
   if (Next_Match == Interrupt)
```

```
        {
           SetupMatchA(Interrupt_Angle); //this sets the start of the pulse
           OnMatchAPinNoChange();
           Next_Match = Pulse;
        }
    else // if Next_Match == Pulse;
        {
           SetupMatchB(Pulse_Start_Angle); //this sets the start of the pulse
           OnMatchBPinHigh()
           SetupMatchA(Pulse_Start_Angle + Pulse_Width); //sets the end of the pulse
           OnMatchAPinLow()
        }
if (MatchB_TransA)
{
    ClearMatchBLatch();
    OnMatchBPinNoChange()
    Next_Match = Interrupt;
}
/* ...continue... */
```

In Example 4, the user wishes to use the match registers to produce a pulse at regular intervals, but interleave the pulses with a channel interrupt. The expected order of the threads is to execute the MatchA on the rising edge of the pulse, followed by MatchB on the falling edge, followed by another MatchA on the interrupt with no pulse. The variable Next_Match toggles between Pulse and Interrupt to keep the states in order. This is a good example of using a single channel to handle multiple, non-conflicting tasks.

The code may operate as expected under most conditions. However, if the Pulse_Width variable is set to a low number, and the speed of the Angle Clock (engine speed) is fast enough, and there are other channels sharing the eTPU, it is possible that after the rising edge MatchB occurs but before the Scheduler can grant the service request, the falling edge MatchA may occur. If this happens, the falling edge would be processed before the rising edge, and the variable Next_Match would not be changed before the pulse is set up again. The result is a rather bizarre operation for one cycle, and since the conditions might be improbable, the bizarre pulse might appear rarely and look like a random event.

There are a number of ways to solve the problem, of course, but one contributing cause was the assumption that the matches would be serviced in order of occurrence. When it is possible that two service requests occur within a short time of each other, the user must plan for the orderly execution of the threads.

In the entry vector tables, the channel conditions MatchA_TransB and MatchB_TransA each have entries, but there are also entries for both conditions together. In the example, the user needs to ensure that when both conditions are satisfied, the MatchB_TransA thread is taken. This can be done in two ways.

The user may populate the entry vector table by explicitly stating the conditions for each vector, for example:

```
else if (MatchA_TransB && !MatchB_TransA)
/*...handles MatchA only...*/
else if (!MatchA_TransB && MatchB_TransA) || (MatchA_TransB && MatchB_TransA)
/*...handles MatchB only and MatchA and MatchB together...*/
```

The same vector encoding can be effected implicitly by reversing the order of the entry vector declarations:

```
else if (MatchB_TransA)
/*...handles all MatchB regardless of Match A...*/
else if (MatchA_TransB
/*...handles the rest of MatchA conditions...*/
```

NOTE

The Byte Craft eTPUC compiler requires that all entry conditions be accounted for in the source code. A convenient way to complete the table for unspecified conditions is to use the trailing *else* to vector to an empty thread containing only an *end* statement. It is important that all link and channel service requests be explicitly called to avoid a condition where an empty thread fails to clear the requesting condition, and a channel hogs all available cycles responding to the uncleared service request.

3 Channel Modes

In the channel action units, the matches and captures can be combined in various ways to control the sequence of events, as well as pin action and service requests. The various combinations are encoded into 13 preprogrammed channel modes which, together with the pin action and time base selections, provide a great amount of flexibility and versatility in the eTPU channel hardware. In some cases, the flexibility simply reduces the amount of software required to support a function, while some channel configurations enable the eTPU hardware to execute timing functions that could not be handled by software.

The modes are summarized in Table 5. The table is organized in sets of columns according to the order in which events can occur in the various modes. The information includes events that are blocked and when they are enabled, and those events that cause a channel service request (shaded). The subsections below walk through each of the modes giving examples where needed. Thereafter, the summary table may be used as a reference when designing an eTPU function.

Channel modes are set up by eTPU software, and can be changed within a function. It is a good idea not to change the channel mode when any event is pending to avoid an unexpected result.

Table 5. Channel Mode Summary

Mode	Initially Blocked	1st Event			2nd Event			3rd Event			4th Event		
		Event type	[blocks] (enables)	Capt.[1]	Event type	[blocks] (enables)	Capt.[1]	Event type	[blocks] (enables)	Capt.[1]	Event type	[blocks] (enables)	Capt.[1]
sm_st	MatchB	MatchA		A&B	TransA		A&B	TransB		B			
		TransA	[MatchA]	A&B	TransB		B						
sm_dt	MatchB	MatchA		A&B	TransA		A	TransB		B			
		TransA		A	MatchA		B	TransB		B			
		TransA		A	TransB	[MatchA]	B						
bm_st	-	MatchA		A	MatchB		B	TransA		A&B	TransB		B
		MatchA		A	TransA	[MatchB]	A&B	TransB		B			
		MatchB		B	MatchA		A	TransA		A&B	TransB		B
		MatchB		B	TransA	[MatchA]	A&B	TransB		B			
		TransA	[matches]	A&B	TransB		B						
bm_dt	-	MatchA		A	MatchB		B	TransA		A	TransB		B
		MatchA		A	TransA		A	MatchB		B	TransB		B
		MatchA		A	TransA		A	TransB	[MatchB]	B			
		MatchB		B	MatchA		A	TransA		A	TransB		B
		MatchB		B	TransA		A	MatchA		-	TransB		B
		MatchB		B	TransA		A	TransB	[MatchA]	B			
		TransA		A	MatchA		-	MatchB		B	TransB		B
		TransA		A	MatchA		-	TransB	[MatchB]	B			
		TransA		A	MatchB		B	TransB	[MatchA]	B			
		TransA		A	TransB	[matches]	B						
em_nb_st	-	MatchA		A	MatchB		B	TransA		A&B	TransB		B
		MatchA		A	TransA	[MatchB]	A&B	TransB		B			
		MatchB		B	MatchA		A	TransA		A&B	TransB		B
		MatchB		B	TransA	[MatchA]	A&B	TransB		B			
		TransA	[matches]	A&B	TransB		B						
em_nb_dt	-	MatchA		A	MatchB		B	TransA		A	TransB		B
		MatchA		A	TransA		A	MatchB		B	TransB		B
		MatchA		A	TransA		A	TransB		B			
		MatchB		B	MatchA		A	TransA		A	TransB		B
		MatchB		B	TransA	[MatchA]	A	TransB		B			
		TransA	[MatchA]	A	MatchB		B	TransB		B			
		TransA	[MatchA]	A	TransB	[MatchB]	B						
em_b_st	-	MatchA	[MatchB]	A&B	TransA		A&B	TransB		B			
		MatchB	[MatchA]	A&B	TransA		A&B	TransB		B			
		TransA	[matches]	A&B	TransB		B						

Table 5. Channel Mode Summary (continued)

Mode	Initially Blocked	1st Event			2nd Event			3rd Event			4th Event		
		Event type	[blocks] (enables)	Capt.[1]	Event type	[blocks] (enables)	Capt.[1]	Event type	[blocks] (enables)	Capt.[1]	Event type	[blocks] (enables)	Capt.[1]
em_b_dt	-	MatchA	[MatchB]	A&B	TransA		A	TransB		B			
		MatchB	[MatchA]	A&B	TransA		A	TransB		B			
		TransA	[MatchA]	A	MatchB		B	TransB		B			
		TransA	[MatchA]	A	TransB	[MatchB]	B						
mB_st	[TransA]	MatchA	(TransA)	A	MatchB		B	TransA		A&B	TransB		B
		MatchA	(TransA)	A	TransA	[MatchB]	A&B	TransB		B			
		MatchA&B	(TransA)	A&B	TransA		A&B	TransB		B			
		MatchB	[MatchA]	B									
mB_dt	[TransA]	MatchA	(TransA)	A	MatchB		B	TransA		A	TransB		B
		MatchA	(TransA)	A	TransA		A	MatchB		B	TransB		B
		MatchA	(TransA)	A	TransA		A	TransB	[MatchB]	B			
		MatchA&B	(TransA)	A&B	TransA		A	TransB		B			
		MatchB	[MatchA]	B									
mB_o_st	[TransA] [MatchB]	MatchA	(MatchB) (TransA)	A	TransA	[MatchB]	A&B	TransB		B			
		MatchA	(MatchB) (TransA)	A	MatchB	[TransA]	B						
mB_o_dt	[TransA] [MatchB]	MatchA	(MatchB) (TransA)	A	MatchB	[TransA]	B						
		MatchA	(MatchB) (TransA)	A	TransA		A	MatchB	[TransB]	B			
		MatchA	(MatchB) (TransA)	A	TransA		A	TransB	[MatchB]	B			
sm_st_e	-	MatchA		A&B	TransA		A&B	TransB		B			
		TransA	[MatchA]	A&B	TransB		B						

Indicates Service Requested

NOTES:

1 In these columns, “A” refers to the TCR selected as the time base for the indicated action in Action Unit A, and “B” refers to the TCR selected as the time base for the indicated action in Action Unit B. Note that A and B selections may be the same or different. Also note that an “A” following a match may not indicate the same TCR as an “A” following a transition in the same line.

3.1 Single Match, Single Transition

The *SingleMatchSingleTransition()* (sm_st) mode is the TPU compatible mode, and should be the first one considered in designing any function. In this mode the MRL in Action Unit B is disabled, and the transition detect latch from Action Unit B is not connected to the service request gate. The mode should be used with IPACB disabled (*DetectBDisable()*). Under these conditions, there will be only two possible channel service requests, one for MatchA and one for TransA, and they have separate vectors in the entry vector table. Also, as was true in the TPU, the transition blocks the match, so if both TDLA and MRLA are asserted, the match certainly preceded the transition or they happened simultaneously.

The mode can be further simplified by disabling either the match or the transition. Since the TPU effectively was either an input or an output, this is the true TPU compatible mode.

NOTE

Matches are disabled by default and can only be enabled by setting the MRLE. This is done as a side effect of writing the ERT to the corresponding Match register, using *EnableMatchA()* or *EnableMatchB()*.

Transition detection is disabled by programming the IPAC using *DetectADisable()* or *DetectBDisa*ble(). This is the default setting.

When an application needs to change the state of an output pin or even just request service from the host at some known time or angle in the future, the single match is all that is needed. If the action is to be repeated, the match will trigger a service request to the eTPU, which can set up a subsequent match. Likewise, if a future pin transition must be detected and the time of the transition captured, this mode is completely adequate.

The sm_st mode should be the first mode considered for new designs. With software support it can handle 85% of all eTPU applications. However, the eTPU enhancements were added primarily to address most of the 15% that were difficult or impossible in the TPU. The rest of the channel modes were each designed with a specific type of application in mind. They should be considered in cases where the simple TPU mode is inadequate, or where the additional features in the channel logic simplify or enhance the features of a function.

3.2 Single Match, Double Transition

The *SingleMatchDoubleTransition()* (sm_dt) mode adds a second transition detection to the previous mode.

NOTE

The only difference between single transition (_st) and double transition (_dt) modes is that a channel service request is made at the first or second transition respectively. In both modes, two transitions can be triggered depending on the IPAC programming. TDLB in _st modes and TDLA in _dt modes will not cause a service request, but the latch conditions will be considered in the entry vector decoding if a service request is made for another reason.

In the eTPU, the transitions are always ordered. The transition detection in Action Unit A must always occur before the transition in Action Unit B can be detected.

Example 5.

```
SingleMatchDoubleTransition();

DetectARisingEdge();

DetectBDisable();

/* This will set TDLA on a pin rising edge but will not cause a service request.

If a subsequent service request is made for a match A, for example, the scheduler will
decode the entry using both MatchA_TransB and MatchB_TransA. */
```

If the application requires the measurement of the width of a pulse, the sm_dt mode will allow both edges of the pulse to be captured before the eTPU engine needs to process anything. This is particularly useful if the user needs to reduce the load on the eTPU engine, or if the pulse width is small with respect to the worst case latency of the eTPU functions. The IPACA can be set to detect the leading edge, and IPACB the trailing edge. The two capture registers will be read into the respective ERT registers, and one subtraction will yield the pulse width.

If Match A is enabled (by writing to the match register) in this mode, a match will also cause a service request. This may be used as an alarm in the event the pulse has not completed by a given time/angle. If the match occurs before the first transition, the capture registers will contain the time bases selected in the action units at the time of the match. During the Time Slot Transition (TST) the capture register values are read into the respective ERTs.

If one transition (necessarily A) has occurred before the match, then capture register A will have the time/angle of the A transition, and capture register B will have the value of the timebase selected for Action Unit B at the time of the match. There is no confusion here because the capture value and the latching of the TDL occur coherently. The interpretation of the values that have been captured is determined uniquely by the states of the TDLs.

If both transitions occur before the match, then the match is blocked. Therefore, if all three latches (MRLA, TDLA, and TDLB) are asserted, this indicates that the second transition occurred after the match.

If the transitions occur after the match request service has begun, the conditions tested by the software and the values in the ERTs will reflect the values latched (coherently) during the time slot transition. If the application requires the latest transition information, the software must rewrite the CHAN register to update the TDLs and the ERTs.

NOTE

During a service routine which had been entered due to a match condition, it is essential to clear the MRL before exiting the thread. If, during that thread, a transition occurs and a TDL is set but not latched into the condition codes, then exiting the thread without clearing the TDL will cause new service request and the transition can then be serviced.

3.3 Both Matches, Single Transition

The *BothMatchSingleTransition()* (bm_st) mode enables the MRLB. It differs from the sm_st mode in that the first match, either A or B, does not request service, but the second match does. In its simplest application, if transitions are disabled, this mode could be used to generate a single output pulse as narrow

as a single TCR time, and to request service after the pulse is complete. It could also be used to switch an output pin without requesting service, if that was required.

As an input mode, *bm_st* might be used to setup a double time-out on an input transition. The matches could include asynchronous TCRs such as an angle and a time, and the MRLs would then indicate if an input event happened after one of the time-out events. If both time-outs occur without a transition, the channel would request a match service. Once the transition occurs, further matches are blocked, so the channel automatically records whether the transition occurs before or after a match.

Note that in a synchronous time-time setup, if the earlier match is set in Action Unit B, the transition would vector to *MatchA_TransB* only when the second match occurs before the transition. If the transition occurs before the first match, the entry vector would be *MatchB_TransA* and MRLB would not be set. And if the transition occurs after the first match (B) and before the second, the entry vector would be *MatchB_TransA*, but MRLB would be set. This gives the user three threads to program separately, e.g. *early*, *on-time*, and *late*. See Figure 2.

In all cases, a transition captures the TCRs indicated for both action units. Therefore, whether the transition is early or on-time, both the time and the angle at the transition is available in the thread. If the second match occurs and no transition is detected, either TCR at the time-out match can be captured.

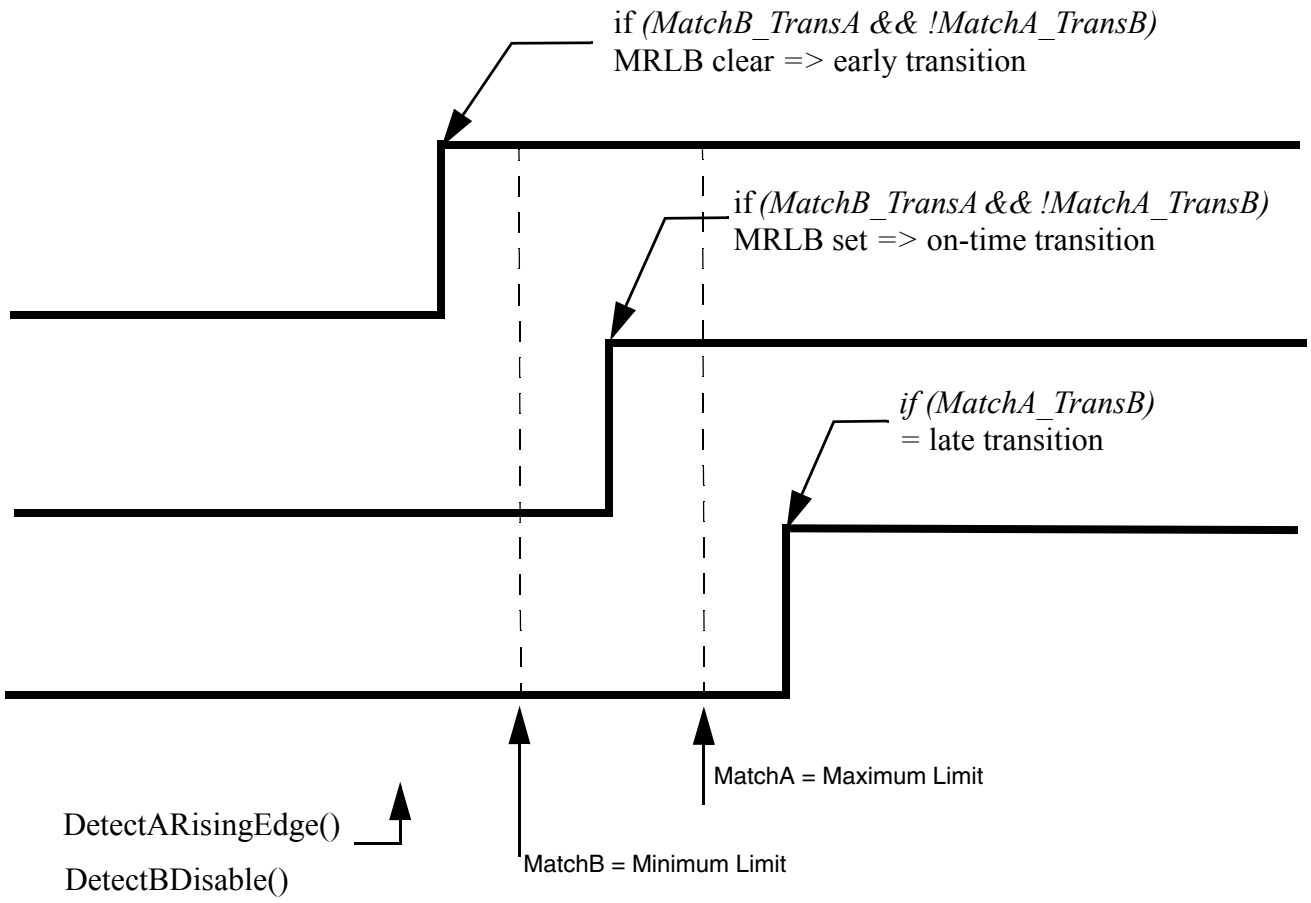

Figure 2. Transition Double Time-out

3.4 Both Matches, Double Transition

In *BothMatchDoubleTransition()* (bm_dt), neither the first match nor the first transition requests service. These can happen in any order, but the first transition will always set TDLA and capture its TCR. Since, as a general rule, a transition capture cannot be overwritten by a subsequent match, the capture register will always have the transition time/angle if TDLA is set.

When the second transition occurs, TDLB will be set and the TCR selected in Action Unit B will be captured. If the second transition occurs before the second match, then it will request service; subsequent matches are blocked, and it is a simple matter to test MRLA to determine if the first match occurred before the second transition.

If both matches occur before the second transition, the second match will request service. Note that after the second match, the second transition (or even both transitions) can occur before the channel is scheduled for service. Again, this condition is easily tested. If TDLB and both MRLs are set, the second transition occurred after the second match. The window testing of a pulse is illustrated in Figure 3

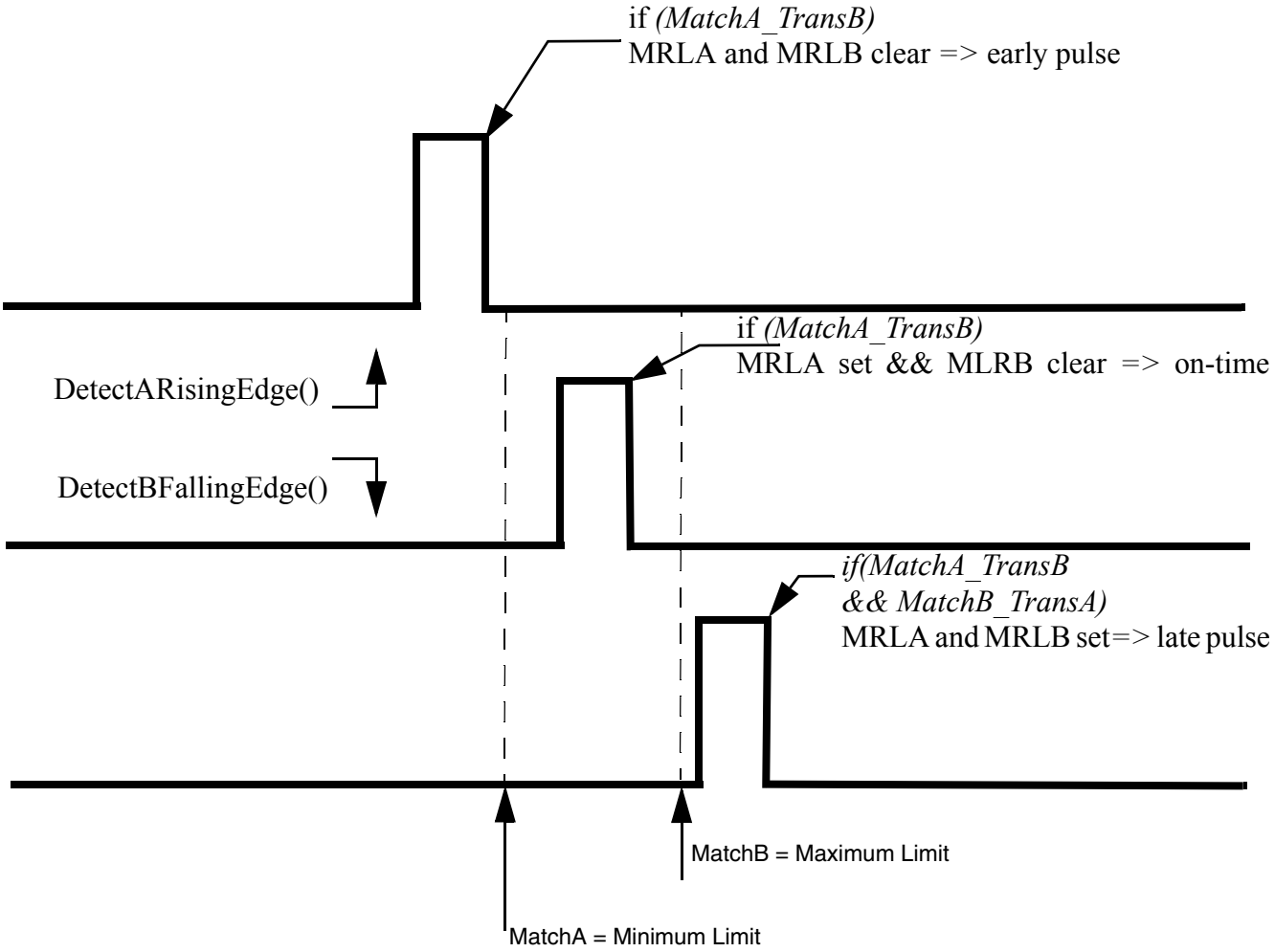

Figure 3. Pulse Time-out

3.5 Either Match, Non-Blocking, Single Transition

In *EitherMatchNonBlockingSingleTransition()* (em_nb_st), the channel behavior is identical to bm_st, except that the first match requests service also. This mode might be used as an output to change the pin and/or request service on the first occurring of a time or angle. For example, if an ignition coil was energized, the application might want to fire a spark at a given angle, but might also have setup a safety "maximum dwell" to fire the spark if the dwell time exceeds a maximum time. This would prevent the coil driver staying on if the engine suddenly stalled.

As an input mode, em_nb_st could be used to check either of two timeouts on an input transition. This would have a particular application when two different asynchronous TCRs were used; for example, time and angle. Use this mode rather than *bm_st* if it is possible that the transition and one match does not occur at all. Note that the second match will also set its MRL and request service as long as it occurs before the transition.

Another use for this mode would be to create an unrelated match event, such as a periodic interrupt timer, sharing a channel with another function. If the channel function is a simple timed output, the second match can be setup with the OPAC set to NoChange. Either match will start a service thread, and the software may handle the combinations as required. Note that if one of the matches is being serviced when the other occurs, as long as the service routine only clears the match that instantiated the service, the other will request service from the scheduler as soon as the first thread is complete.

If this mode is used as an input with an unrelated time-out match, it is possible that the input transition blocks the unrelated match. In that case, as long as the transition service routine does not alter the match and the match is set for greater-or-equal comparison, the match will recur after the TDL is cleared and the transition thread is exited.

As with other _st modes, a second transition can be detected by this mode if the IPACB is set. However the second transition will not request service. This could be useful for an application to detect a fast noise pulse which could set a rising and falling TDL in short order. If the second transition is detected, the TCR in Action Unit B will be captured at the time of the second transition.

3.6 Either Match, Non-Blocking, Double Transition

In the EitherMatchNonBlockingDoubleTransition() (em_nb_dt) mode, the first transition does not request service, but each transition blocks its respective match. In an input function, this mode would provide independent timeouts for both edges of an input pulse. Since the first transition must always precede the second transition, the matches are presumably similarly ordered in the setup.

The difference between this mode and the bm_dt mode is that this one can be used to establish two *late* timeouts, one for each edge, while the other has an *early* and a *late* time-out for one of the edges. See Figure 4. The thread decoding is shown in Table 6. Note that in some cases, a TDL may be set after the thread requests service but before the TST occurs.

Note that if the IPACs are both set for the same edge, this mode could be used to provide timeouts for two consecutive pulses, whose edges are as close as one TCR time.

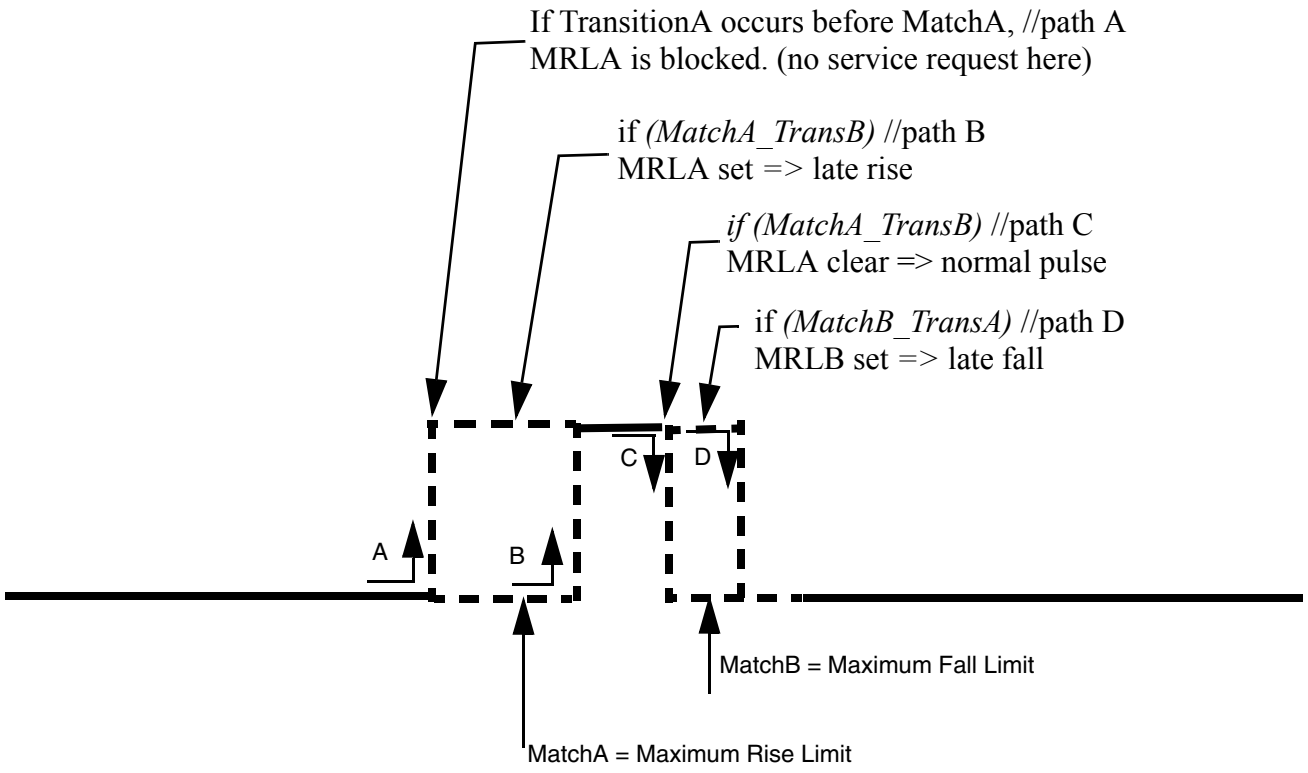

Table 6. Thread Decoding

Path	Service Request	S.R. Time	MRLA	MRLB	TDLA	TDLB
A-C	*MatchA_TransB*	C = TransB	clear	clear	set	set
A-D	*MatchB_TransA* [&& *MatchA_TransB*]	MatchB	clear	set	set	clear/set*
B-C	*MatchA_TransB*	MatchA	set	clear	clear/set*	clear/set*
B-D	*MatchA_TransB* [&& *MatchB Trans A*]	MatchA	set	clear/set*	clear/set*	clear/set*

* Clear at service request, but may be set at service time, depending on timing.

Figure 4. Two Edge Timeouts of a Pulse

3.7 Either Match, Blocking, Single Transition

In the *EitherMatchBlockingSingleTransition()* (em_b_st) mode, the first match to be satisfied will block the other and will generate a service request. The configuration of the channel logic when this mode is selected is shown in Figure 5. Note that the matches are blocked by setting the MRLEs; therefore, if and only if both matches occur simultaneously, both MRLs will be set.

This mode is particularly useful when the matches are setup to compare to asynchronous TCRs. For example, one match might be satisfied by an angle and the other by a time. The MRLs will unambiguously

indicate which match occurred first. Whenever a match occurs, it captures the TCRs indicated by both action units.

If IPACA is programmed to detect an input pin transition, that transition will block the matches if they had not yet occurred, capture (and overwrite if matches have occurred) the TCRs indicated by both action units, and request service.

As in other _st modes, a second transition can be detected by Action Unit B, but the detection will not request service. To avoid ambiguity if detection of an input transition is not required, the user should program the IPACs to *NoDetect*.

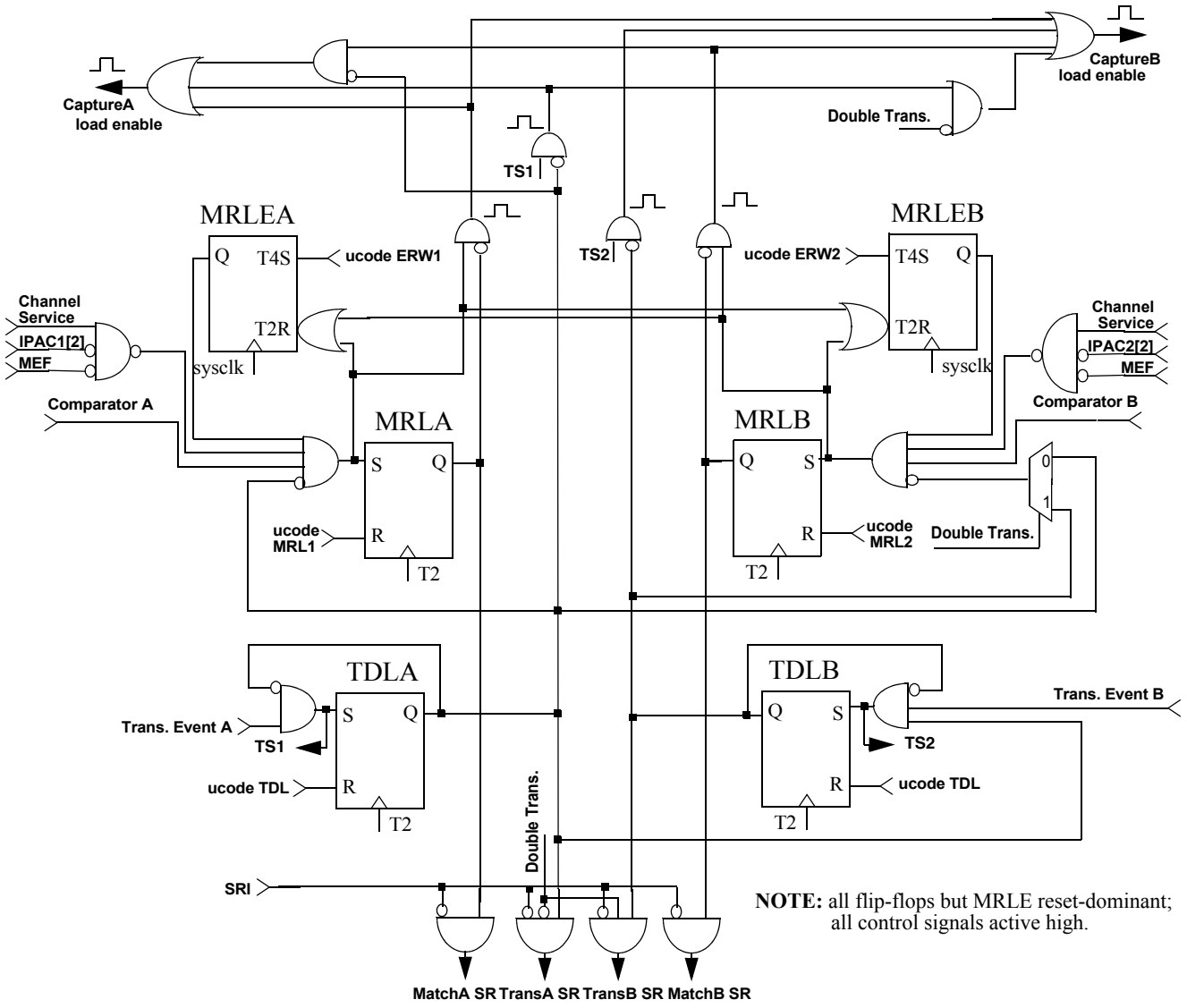

Figure 5. Either Match, Blocking Modes (em_b_st, em_b_dt)

3.8 Either Match, Blocking, Double Transition

In the *EitherMatchNonBlockingDoubleTransition()* (em_b_dt) mode, each transition will block its corresponding match, and only the first match and second transition will request service. The matches block each other as in the em_b_st mode, so that any MRL set indicates a time-out on the corresponding transition. An MRL before the first transition will capture both TCRs and request service, but subsequent transitions may overwrite the corresponding capture registers. Match B after the first transition will set MRLB and capture the TCR in Action Unit B. A subsequent Transition B will capture the selected TCR at the later time. If Transition A is followed by Transition B without a match, both MRLs will be blocked and the TCRs will be captured on both edges.

The only difference between this mode and em_nb_dt is the mutual blocking of the matches. If Action Unit A and Action Unit B are comparing to different TCRs, the MRL that is set will unambiguously indicate which match (time-out) occurred first.

3.9 Match B, Single Transition

In the *Match2SingleTransition()* (m2_st) mode, transitions are initially blocked. Transition A is enabled by Match A. If Match B occurs before Match A, it blocks Match A and the transitions and it requests service. Otherwise, Transition A can be detected and will capture the TCRs in both action units, block a subsequent MatchB, and request service. Transition B can be detected after Transition A; it will update the capture of the TCR in Action Unit B, but will not issue a service request.

In an output function, this mode can drive a pin from Match A on the condition that a match on Match B has not occurred. For example a fuel injector may be set to open at a given angle under the condition that a time-out has not expired, which might indicate that the engine has stalled.

In an input function, Transition A is expected to occur between Match A and Match B. If that happens, Match B is blocked, the transition is serviced, and the next transition can be setup. If the transition occurs before Match A, it will not be detected. Then, when Match B occurs, the error can be detected by the fact that MRLA is set. If Match B occurs and MRLA is not set, then Match B preceded Match A and no transitions can be detected. Transition B, if set up, can occur after any Transition A. It will set the TDLB, but not initiate a service request.

Example 6.

The *Match2SingleTransition()* mode may be used in an application where a string of regular input pulses is expected and the edges of the pulses could be characterized by noise too slow to be filtered out by the pin filters. For example, if the application is tracking a toothed wheel on a rotating shaft with a variable reluctance sensor, the sensing circuit may operate well at moderate to fast shaft speeds but exhibit hashy edges when the rotation is slow, due to the low signal level from the VR sensor.

A driver interfaced to such a system might detect and process the first transition of the pin input signal, but because of the speed of the eTPU, continuing noise on the input pin might be interpreted as a subsequent tooth edge. To reject this noise, the software could setup a match time after the first edge is detected during which subsequent transitions are ignored. This

blocking time could be a fixed delay or a variable driven by the measured speed of the shaft. The second match could then be setup to request service if the following transition does not occur within a selected time, indicating possible stalling of the shaft.

If this technique is used to reject a noisy edge, it may be necessary to process both the rising and the falling edges of the input signal, as noise pulses on a falling edge may also be detected by a *DetectARisingEdge()* transition.

3.10 Match B, Double Transition

In the *Match2DoubleTransition()* (m2_dt) mode, Transition A does not block matches nor issue a service request. These actions occur on Transition B. Otherwise the action is identical to the previous mode. If Match B is set up to occur after Match A, a pulse detection on the channel will be detected, as shown in Figure 6. Note that if Match B occurs before Match A, then only MatchB_TransA will request service, and only MRLB will be set.

This mode has an obvious application in qualifying an input pulse with a lower and upper limit. A less obvious application would be on a single transition time-out, where the service needs to be delayed until some time after the transition. If DetectBDisable() is selected, the Match A could set the blanking time for an input transition, and the transition time could be captured without a service request. The Match B would then be setup to request a service thread during which the software could test TDLA to find out if the transition happened at all. This might be used where the software only needs to confirm that something expected actually did happen.

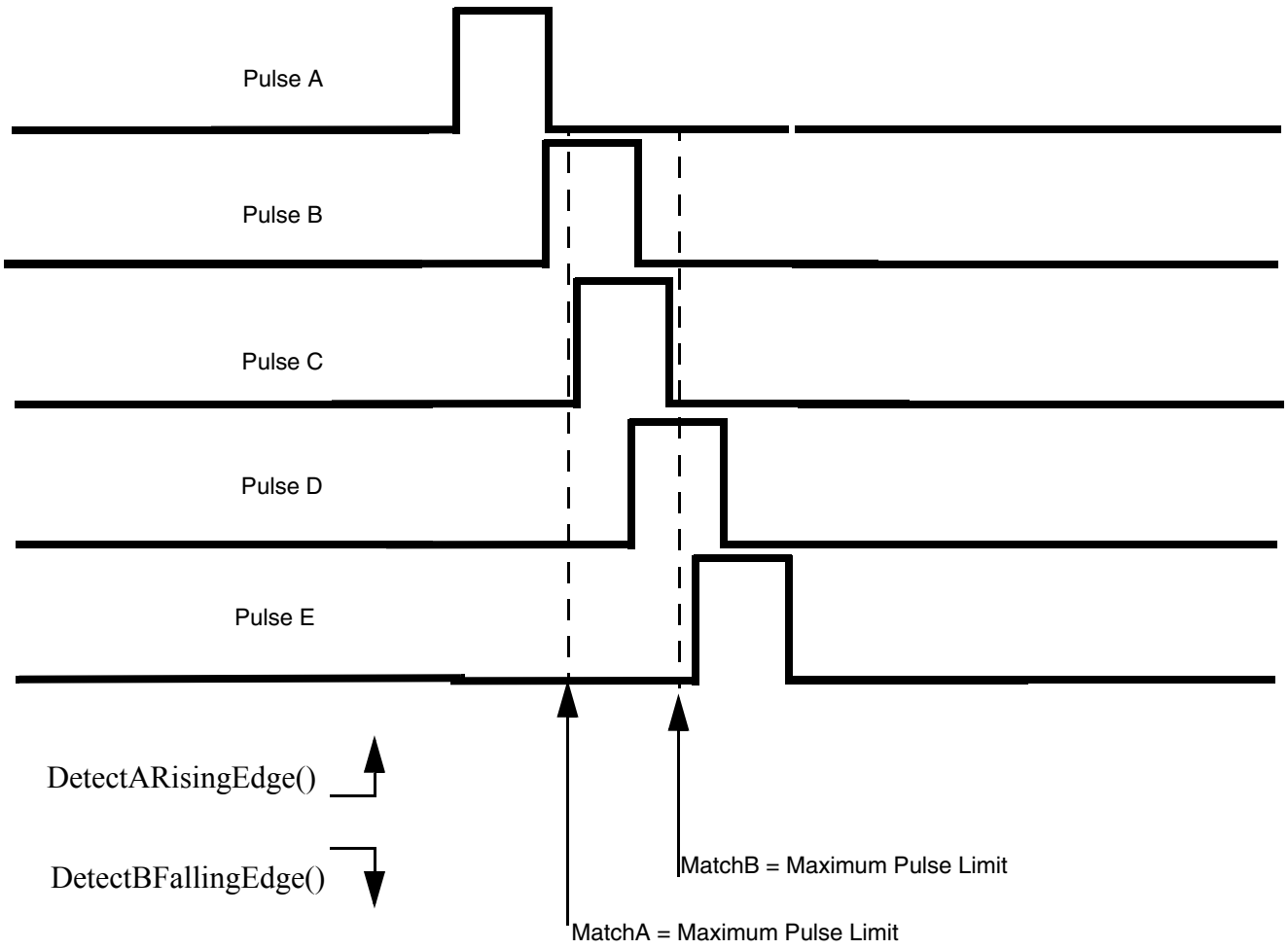

Table 7. Thread Decoding

Pulse[1]	Service Request Vectors	SR Time	MRLA	MRLB	TDLA	TDLB
A	*MatchA_TransB && MatchB_TransA*	MatchB	set	set	clear	clear
B	*MatchA_TransB && MatchB_TransA*	MatchB	set	set	clear	clear
C	*MatchA_TransB && MatchB_TransA*	TransB	set	clear	set	set
D	*MatchA_TransB && MatchB_TransA*	MatchB	set	set	set	clear/set
E	*MatchA_TransB && MatchB_TransA*	MatchB	set	set	clear/set	clear/set

NOTES:
1 A and B cannot be distinguished. D and E can be distinguished by the Capture values.

Figure 6. Match B, Double Transition Input Detection

3.11 Ordered Matches, Single Transition

In the *Match2OrderedSingleBlockingTransition()* (m2_o_st) mode, Match A must occur before any other channel event. It does not generate a service request, but the MRLA flag will be set. Thereafter, either Match B or Transition A may occur and will request service, but if Transition A occurs first, it blocks a subsequent Match B. Transition B may follow Transition A, and will set TDLB but will not request service.

This and the following mode are the only ones where the matches are ordered, and they can be useful to check elapsed time against elapsed angle in a rotating machine. In addition, if the match registers are set to the same time, and greater-or-equal matches are selected, the channel can produce a pulse exactly one microcycle wide.

In an input function, the matches provide an absolute acceptance window for an input transition. If TDLA is set, the transition occurred after Match A and before Match B.

3.12 Ordered Matches, Double Transition

In the *Match2OrderedDoubleTransition()* (m2_o_dt) mode, Match A must occur before any other channel action is enabled. Thereafter a pulse (double transition) may be detected, and the second transition (B) will request service. If the Match B occurs before the second transition, it will block subsequent transitions and issue a service request.

The ordered matches differ from the *Match2DoubleTransition()* mode in that the TDLs will not be set after the Match B. This takes some ambiguity out of selecting the service thread, as shown in Table 8.

Table 8. Ordered Matches, Double Transition (See Figure 6)

Pulse[1]	Service Request Vectors	SR Time	MRLA	MRLB	TDLA	TDLB
A	MatchA_TransB && MatchB_TransA	MatchB	set	set	clear	clear
B	MatchA_TransB && MatchB_TransA	MatchB	set	set	clear	clear
C	MatchA_TransB && MatchB_TransA	TransB	set	clear	set	set
D	MatchA_TransB && MatchB_TransA	MatchB	set	set	set	clear
E	MatchA_TransB && MatchB_TransA	MatchB	set	set	clear	clear

NOTES:
1 A and B cannot be distinguished.

3.13 Single Match, Single Transition, Enhanced

In *SingleMatchSingleTransitionEnhanced()* (sm_st_e) mode, the Capture B register continually captures the time of the IPACA transitions on the input of the channel digital filter. When the filter conditions are satisfied and the signal passes through the filter, the TDLA is set and Capture A records the time of the

qualified transition. Subtracting ERTB from ERTA in the service thread will yield the lag time through the filter. This is a special mode used to check the action of the channel filter. Please see the Reference Manual for details.

4 Using the Channel Hardware

Because of the complex design of the channel hardware, the engineer may have a choice of several approaches to meet the requirements of the design. In this section we will try to give some guidance on how to approach a design problem using the eTPu in the most efficient manner.

The first question that arises is how to divide the task into hardware, eTPU software and host software partitions. There are two competing demands on the system which must be considered in order to make a partitioning decision. The suitability of a real time controller system can be measured in timeliness and flexibility. The engineer chooses an MCU to control a system principally for the flexibility that the software offers. But software execution has a finite latency which might exceed the timeliness requirements for some control systems. Timeliness is not the same as speed. If the temperature of a house drops below the thermostat set point, it makes little difference if the relay closes in 10 milliseconds or 500 milliseconds. The systems engineer needs to place realistic requirements on the performance of a system.

In the eTPU, one engine is servicing as many as 32 channels. In most applications, the servicing of a channel event takes less than one microsecond, and events may be milliseconds apart. The worst case loading on the eTPU engine in a Reference Design was found to be less than 10% under the worst conditions. However, a too complex algorithm in the service routine for one channel can dominate the engine and rob performance from the other channels. This is the best argument for migrating as much as possible of the control algorithm to the host software.

Partitioning between the channel hardware and eTPU software is not as clear. Moving as much as possible to the hardware will reduce the load and latency on the eTPU engine, but the hardware is the least flexible of all the subsystems. For example, an input function measuring the width of a pulse might hang up a system when the second edge of the pulse does not occur.

Because of the speed of the eTPU engine, the best approach to selecting a channel mode is to stay as simple as possible unless performance demands more hardware complexity.

4.1 Selecting the Channel Mode

The simplest channel mode is sm_st. This mode can handle more than 85% of applications, as has been shown by the success of the TPU. The information below can provide a guide when there is a reason to change the channel hardware configuration.

4.1.1 Understanding the Channel Modes

The thirteen channel modes are combinations of subsystem configurations of the channel hardware. The subfields that make up the notation for the modes are listed with their interpretation in Table 9.

Table 9. Channel Mode Notation

Selection	Configuration	Interpretation
sm	SingleMatch...	Only Match A is enabled.
bm	BothMatches...	Both Matches are enabled in any order and the second one will request service.
em	EitherMatch...	Both Matches are enabled in any order and each one will request service.
m2	MatchB...	Both Matches are enabled, but only Match B will request service. Transition A is disabled until Match A occurs. Match B occurring before Match A will block Match A
_b	...Blocking...	Matches mutually block each other.
_nb	...NonBlocking...	Matches do not block each other.
_o	...Ordered...	Match B is blocked until Match A occurs.
_st	...SingleTransition	Transition A will request service and block subsequent Matches
_dt	...DoubleTransition	Transition B will request service and block Match B In bm mode, Transition B blocks both Matches.

In addition to the rules listed in the table above, the following rules always apply:

1. Transitions are always ordered. Transition B is enabled by Transition A.
2. Transitions will always be detected if the pin transitions according to the corresponding IPAC.
3. Detection of a transition will always set the corresponding TDL.
4. In a DoubleTransition (_dt) mode, if the IPACB is NoDetect, then the transition will not request service.
5. Channel action flags, MRLs and TDLs, are latched into the eTPU during the Time Slot Transition to a new thread. Flags that are set after that time can be read by rewriting the CHAN register, or on a subsequent channel service.
6. For resolution of the entry vectors, all channel latches are considered, not just the ones that requested service.
7. Each match captures the TCR selected for its corresponding Action Unit. In a Blocking (_b) mode, the first match captures both Action Unit TCRs.
8. Transitions always capture their corresponding TCR.
9. In a SingleTransition (_st) mode, Transition A captures TCRs selected for both Action Units, but a subsequent Transition B will re-capture the TCR indicated for Action Unit B.
10. Captures resulting from matches never overwrite captures resulting from Transitions.
11. Captures resulting from Transitions always overwrite capture resulting from matches.
12. Channel action flags and capture register values are always coherent in the eTPU.

4.1.2 Using Double Matches

The double match registers are required where there is a chance that a second pin action will be required before the first one is serviced. For example, a PWM that can be modulated continuously from 0% to 100% duty cycle may require both transitions on the pin before the first can be serviced.

Another important use for a double match is when two asynchronous timebases are used, such as time and angle. A single pin action can be programmed into both OPACs and the first occurring will switch the pin. The service request can be made on the first or the second that occurs, or on a specific match.

The two matches may not even be related. One could be used to switch the output pin, and the other as a wake-up call for an unrelated function. Or one could be used to qualify a transition and the other for an unrelated function.

Finally, there are cases where both matches have identical OPACs and are working with the same TCR. The application may calculate the operations separately, and the separate action units would simply be a means to simplify the software.

4.1.3 Using Double Transitions

The most obvious need for the DoubleTransition modes is to detect two transitions that occur too close together for the latency of the system. For example, a pulse width can be measured to the accuracy of one TCR count. However, there are other situations where the double transition circuitry in the eTPU channels can provide significant functionality.

If a pulse edge is noisy, a single capture register will detect the first programmed transition of the pin and record its time. If two transitions are programmed in a SingleTransition mode, the presence and timing of a second edge can be recorded. This can be done by programing the second IPAC to the same or opposite sense of IPACA.

Sometimes a pin transition needs to be recorded, but the edge need not be serviced until some later event. In that situation, a DoubleTransition mode can be selected and IPACB can be programmed to NoDetect. The transition will be recorded and the timebase captured, but the channel will not be serviced as a result. A subsequent match, or even an event on another channel, can be used to service the detected transition. Remember that even though the transition did not request service, its TDL will be used to resolve the thread vector.

4.1.4 Combining Matches and Transitions

In the case of a noisy input edge, the first transition may be detected and processed and the channel may be setup for the following transition much faster than the noise on the first transition settles out. The most common way to handle this noise is to block out a subsequent transition for a selected time after the first one is first detected. By using a MatchB (m2) mode, Transition A is blocked until Match A occurs. The software may setup Match A to be a fixed time or a time derived from the expected period of the input transitions.

The operation can be carried a step further, using the Ordered (_o) modes. In these modes, the matches establish an acceptance window for the transition. In other modes, the second transition may not be blocked, but service is requested on the first or second match.

In MCUs where the input and output pin are separately connected, one channel can indeed do an input function and output function simultaneously. A particular case of this arises in some MCUs where the analog to digital converter requires a time or angle trigger signal. It is possible to drive the ADC trigger

with the output match function of an eTPU channel, and use the input side of the same channel for an unrelated function.

4.1.5 Interactive Pin Actions

The eTPU channel structure provides output pin actions that are directly keyed to input pin states, providing interaction without software delay. These OPAC states may be useful to provide safety cutoff of an output drive when a feedback signal is found to be at an incorrect state a certain time after the drive is applied.

In Figure 7, the output pin drives a load through a high side driver transistor. When the gate voltage is applied, the load voltage will be low for a time due to the turn on delay of the transistor. After a short time, if the load is healthy, the voltage is expected to go high. If the channel input pin is connected to the load, it can be sampled after a short delay and the channel output can be turned off if the load voltage is not high, indicating a possible shorted load. Once the software is set up, the programming of the load drive is simple, as can be seen in Example 7.

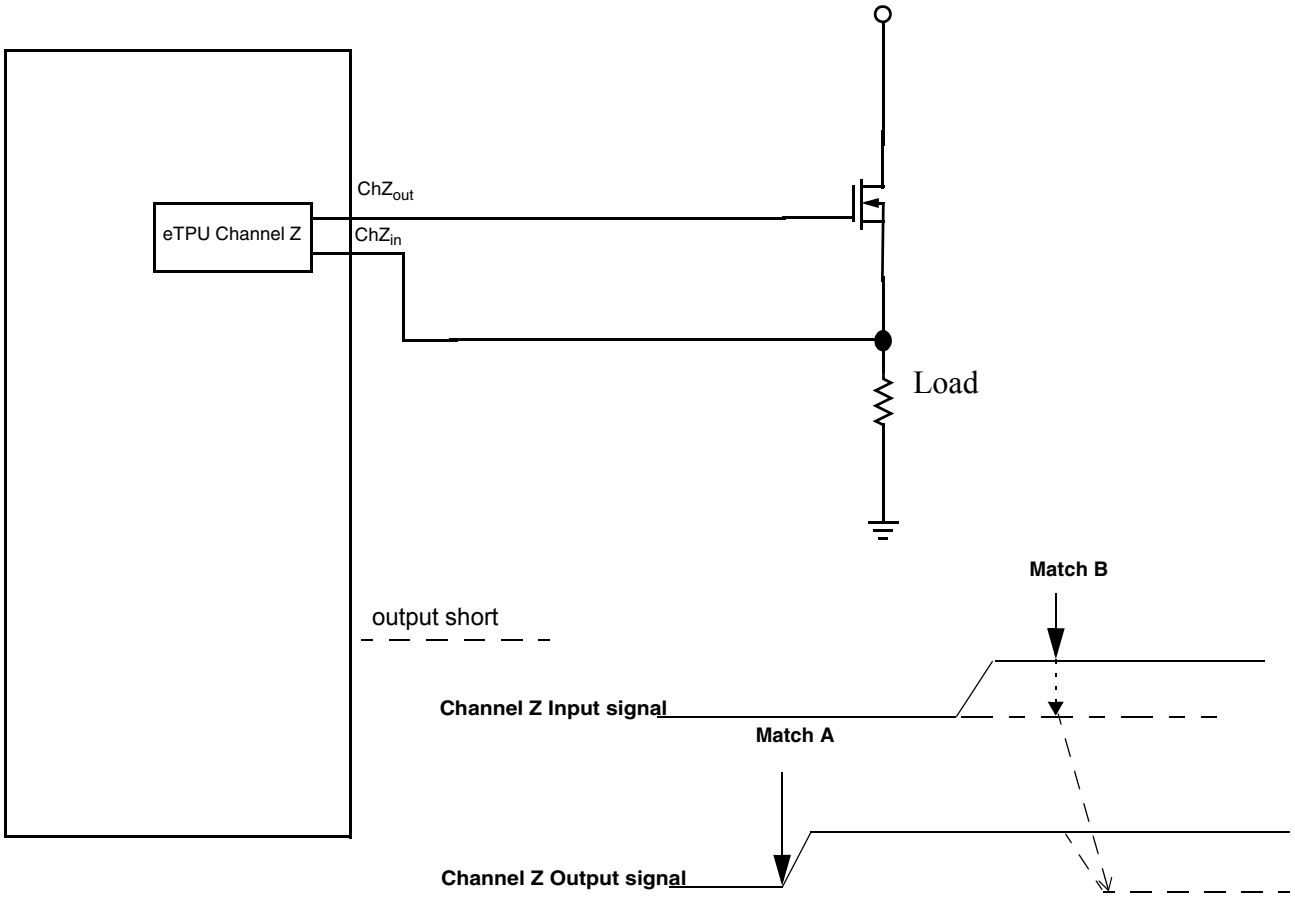

Figure 7. Load Sense

Example 7.

```
/* The following drives a load through a high side driver and tests for shorts */
/* ... */
/* ... In the setup routine... */
BothMatchesDoubleTransition() //
TimeBaseAMatchTcr1CaptureTcr1GreaterEqual()
OnMatchAPinHigh() // OPACA - Main Drive Signal
DetectALowOnMatchA() // IPACA - Load Voltage is low.
TimeBaseBMatchTcr1CaptureTcr1GreaterEqual()
DetectBLowOnMatchB() // IPACA - Is Voltage Low?
OnInputActionBPinLow() // OPACB - Shut off drive
/* ... */
/* The following sets up the drive to turn on the load at time Load_Start_Time and the
feedback test to occur a short time later*/
SetupMatchA(Load_Start_Time); //turn on load at a specified time
SetupMatchB(Load_Start_Time + Transistor_Delay); // sets up feedback test
/* ... */
```

In another example, the eTPU might be applied to drive an inductive solenoid, perhaps a valve, as in the circuit in Figure 8. When the load is turned on, the current will build according to the L/R time constant. If the load is shorted, current will build instantly and perhaps be limited only by a small current sense resistor. Continued application of the drive may damage the output transistor.

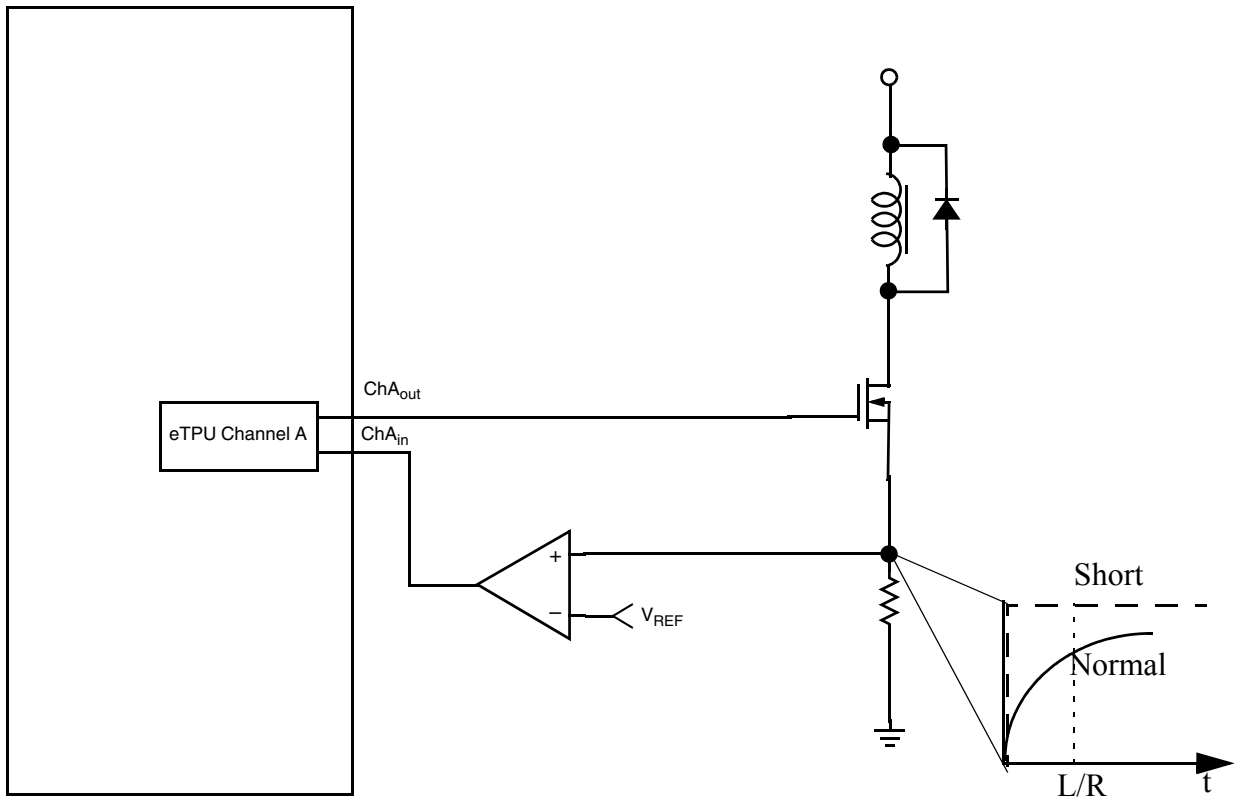

Figure 8. Driving an Inductive Load

In the circuit, we can feed back the current sense to the input pin of the eTPU channel that drives the output. The channel hardware can be used to provide the required circuit protection.

Example 8.

```
/* The following drives an output inductive load and tests for shorts */
/* ... */
/* ... In the setup routine... */
EitherMatchNonBlockingSingleTransition() //
TimeBaseBMatchTcr1CaptureTcr1GreaterEqual()
OnMatchBPinHigh() // OPACB - Main Drive Signal
TimeBaseAMatchTcr1CaptureTcr1GreaterEqual()
DetectAHighOnMatchA() // IPACA - Is Current High?
OnInputActionAPinLow() // OPACA - Shut off drive
/* ... */
```

```
/* The following sets up the drive to turn on the load at time Pulse_Start_Time and the
feedback test to occur a short time later*/
SetupMatchB(Pulse_Start_Time); //start a pulse at a specified time
/* L/R is, say, 0.1 msec. Do the short circuit test at 1 microsec. */
SetupMatchA(Pulse_Start_Time + One_Microsecond); //this sets current level test
/* ... */
/* Match Handling */
if (MatchA_TransB) // In a _st mode, this can only be entered if MRLA is set
{
  if (TransitionALatch)
    {
     Shorted_Load:
     /* Handle load short here */
    }
  else
    {
     /* Handle normal pulse operation here */
    }
  /* ... */
ClearMatchAEvent();
ClearMatchBEvent();
ClearTransitionEvents();
}
if (MatchB_TransA && !MatchA_TransB)
{
ClearMatchBEvent();
}
/* ... more... */
```

In Example 8, the eTPU is set up to turn on an inductive load at a specified time. A current sensing resistor will indicate the exponential rise of current in the circuit after the transistor is switched on. When the load is healthy, the current in the resistor will rise according to the L/R time constant. This may be in the range of hundreds of microseconds. If the load is shorted, the current will rise very fast, depending on the turn-on time of the transistor. When the current sense voltage is fed back to the input of the eTPU channel, it can be tested against a reference as a specified time after the load is applied.

With a normal load, the voltage is turned on at the time set by Match B. The current will rise in the resistor according tho the applied voltage and the L/R time constant. When Match A tests the comparator output

a short time after the application of the drive, the resistor voltage will be well below the threshold and the comparator will be off. The match conditions will not be satisfied.

When the load is shorted, the current will rise very fast, and at the Match A time, the comparator output will be high. MRLA will be set, and since the OPACA is set to turn off the drive when Match A occurs, the drive will be safely turned off. There is no software latency in this channel action, but a unique channel entry condition will be satisfied when the shutdown occurs, allowing software error detection and recovery.

A similar scheme may be used to detect a shorted resistive load with a high-side drive, accounting for the turn-on delay of the transistor.

5 Channel Service Request

By setting up the channel hardware for optimal handling of the input and output pin events, the application can ensure that the timing of these events is as accurate as the MCU clock, regardless of the software load on the eTPU, as long as the total load is reasonably small. Once a transition is captured or a match completed, the software required to record the event and set up the next is usually straightforward. If the user avoids partitioning too much of the control algorithm into the eTPU software, there are only a few additional matters to consider.

5.1 Servicing Order

In Example 8, the normal channel operation will set both the MRLA and the MRLB. Abnormal operation will set TDLA. Even if the delay between the matches is as little as 1 microsecond, in most cases the eTPU will respond to the MRLB before the MRLA is asserted. Since we are not sure of the health of the load at this time, we would not want to setup the next match yet. However, we must service the request and clear the MRLB at least, or we will get continual MatchB service requests until we do.

It is also possible that the fault detection occurs before we schedule the channel for service. Thus in the MatchB service routine, if TDLA is set, we should clear it and service the fault condition.

Note, however, that in the example whenever TDLA is set, MRLA must also be set. If we give MatchA_TransB priority, we can handle the fault condition in all those cases where the scheduler response was slow. Since MatchA indicates that the pulse is completed, if we do not also have TDLA, we have a normal event. So what do we do with MatchB? Just clear the flag and handle everything after MatchA.

The eTPUC compiler will automatically give priority to the first listed channel condition. Thus if the source code uses the condition MatchB_TransA followed by MatchA_TransB, the first service routing will be entered whenever MRLB or TDLA are set, regardless of MRLA and TDLB. Of course the priority can be explicitly stated by using the following conditions (in either order):

Example 9.

```
if ((MatchB_TransA && !MatchA_TransB) || (MatchB_TransA && MatchA_TransB))
 /* Handles MRLB OR TDLA regardless of MRLA or TDLB */
  {
    /*...*/
  }
/* ... */
else if (!MatchB_TransA && MatchA_TransB)
/* Handles MRLA or TDLB if MRLB and TDLA are clear */
  {
    /*...*/
  }
/* ... */
```

The following hints should be considered when deciding on channel service priority:

1. Give priority to the exceptional cases. Normal operation may be contra indicated.
2. Do not assume that later matches will be serviced later. Remember that latency might delay the scheduling of a service routine until a subsequent channel event occurs. Both events will be equally considered by the scheduler.
3. Remember the flags that did not request service. They will also be considered when the scheduler sorts out the service thread.
4. Consider using the channel flags to change priorities at run time.

5.2 Clearing the Channel Latches

In general, all channel latches should be cleared during the service routine they called. There are exceptions to this rule. In some cases when two channel latches (such as MRLA and TDLB) are requesting service, the service routing may test for one latch condition, service it, clear that condition, and exit. The second condition will cause the channel to be rescheduled and the routine entered a second time. This strategy might be used where the conditions are unrelated (for example, an input pin transition and a simultaneous time-out of a wake up timer). The system may call for high priority handling of one condition and low priority handling of the other. If the low priority handling routine is lengthy, it might improve performance to handle the high priority thread, clear the corresponding latch, and exit the service routine to enable the eTPU to handle other channels that might be requesting service before the scheduler responds to the uncleared latch for the low priority thread.

It is a good idea to also clear those channel latches that “cannot” be set. Even in sm_st, it is possible that host action caused a condition which set MRLB or TDLB. The microcode generated for clearing the channel latches is a single instruction, regardless how many channel latches are cleared, as long as the source instructions are grouped together.

An uncleared channel latch may continually request service. This may not block any other channel from being serviced, but it will consume all the idle cycles of the eTPU. Depending on the priority of the channel, this will have some effect on the latency of all other channels requesting service.

5.3 Timing the Channel Latch Clearing

In many situations, the channel service routine will calculate a match at a time which is in the past. This would be an artifact of the algorithm driving the channel and might be perfectly acceptable for the system. However, in this case, care must be taken in the timing of the match service routine to ensure that the MRL is cleared at the proper time.

The eTPU enables a match by setting the MRLE when the ERT is written to the match register. The MRL is cleared by a specific command. If the match is enabled too early, a past due match will be satisfied, and the MRLE will be cleared. Then when the corresponding MRL is cleared, the new match will be lost.

On the other hand, if the MRL is cleared too early, it is possible that it will be set again before the match register was written. This is an unlikely situation, since if the MRL was set, the MRLE should be cleared. However, improper coding in a handling routine can give rise to the situation.

If matches were enabled, and a match occurs simultaneously with the rewriting of the match register, the “old” match will be recognized, MRL will be set again, and the MRLE will also be set, enabling a subsequent new match on the new value.

If the MRL is cleared in the same instruction as the new match value is written, the MRL cannot be subsequently set except by matching the new register value. This is the preferred coding, and it can be effected by grouping the *WriteErtAToMatchAAndEnable()* with the *ClearMatchALatch()* instructions in the source code.

Matches can be disabled by the microcode in two different ways--note the differences. In the entry vector of each thread is a one-bit field which allows the user to disable all matches during the duration of the thread. This option in the entry vector is evoked by including *DisableMatchesInThread()* or alternately *match_disable()* at the beginning of the thread microcode sequence. This operation blocks the signal from the MRL, so any temporary match conditions that are cleared before the end of the thread will not set the MRL.

The thread can also assert *DisableMatchDetection()* during the thread. This clears both the MRLEs, and the operation is not negated simply by ending the thread. Each MRLE that is cleared must be subsequently re-enabled by *WriteErtAToMatchAAndEnable()* or *WriteErtBToMatchBAndEnable()*. One caution should be raised here. The setting of the MRLE also writes the ERT to the match register. The desired match value may have to be read and saved before other operations are performed.

The TDL is set only by the transition to the selected level or by sampling the pin at a match compare, not by the level itself. A transition set on the rising edge and subsequently cleared will not be set again until the pin falls and rises again. If the instruction negating TDL occurs simultaneously with the transition, the TDL will be negated, but the dependent captures and/or pin actions will occur.

Finally, there is no microinstruction that tests and clears the MRLs or TDLs simultaneously, so there can be no ambiguity introduced by compiler instruction packing.

5.4 Dynamically Changing the Channel Configuration

In general it is not a good idea to change the channel mode once the channel is running. When this is necessary, it is important to ensure that MRLs and TDL are not set during the transition. Also it is important to note that match conditions that are blocked by one mode might be enabled with the mode change.

When all of these precautions are ignored, it is probably best to refer to the Reference Manual and compare the schematics (for example, Figure 5) of the two modes to determine the possible outcomes of the change.

6 Summary

The channel logic in the Time Processor Unit (TPU) could be setup in 32 different configurations (8 TBS values times 4 PAC choices). If we consider only the modes, IPACS and OPACS, and ActionUnit selections, there are 43,680 different configurations of the eTPU channels. This application note attempted to simplify the process of making the selections to help the user select an optimal configuration for the requirements.

This note has been double checked against the Reference Manual and the original eTPU Block Guide, and any errors that were found have been corrected. If the user detects any inconsistency, or if this note appears confusing in any way, please contact the author.

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

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

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

AN2933
Rev. 0
12/2004