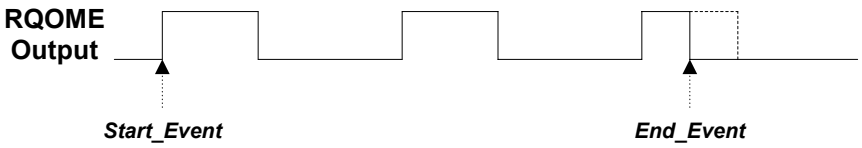
eTPU Function Implementation:

Repeating Queued Output Match



MCD – 5406

Revision 7.1

June 13, 2005

DELPHI

CONFIDENTIAL

Microcode Technology
Powertrain Electronics Software Technology
M/S CT-40D, Delphi E & S, Kokomo, Indiana 46904-9005

This page intentionally left blank

Table of Contents

This page intentionally left blank

eTPU Function:
Repeating Queued Output Match

1. Introduction

The Repeating Queued Output Match (RQOME) algorithm is designed to allow the user to set up a sequence of output match events. Each event may be specified individually as either an absolute event or as a relative offset to a previous event. The user programs the time base and the output pin state for each event and also determines whether or not each event will issue an interrupt and/or a DMA trigger to the host CPU. Repeating loops may be set up within the sequence, and the entire sequence may be programmed to execute only once or continuously.

The first event of a sequence may be programmed by the user or else determined from the last event of a previous sequence. The user also has the option to terminate a sequence at a set time (or angle – see below) regardless of which portion of the sequence is operating at the time.

If desired, the RQOME primitive may be set up to record the actual time/angle of each event in units selectable by the user.

If the eTPU's angle clock feature is used, then TCR2 will represent an angular value, while TCR1 represents a time value. If the angle clock feature is not used, then both TCR1 and TCR2 represent time values.

2. System Overview

2.1 Inputs

There are no inputs required for the operation of this function.

2.2 Outputs

There is one output produced by the operation of this function (see **Figure 1**).

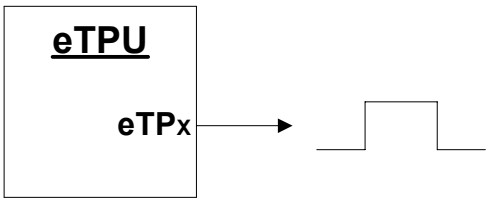
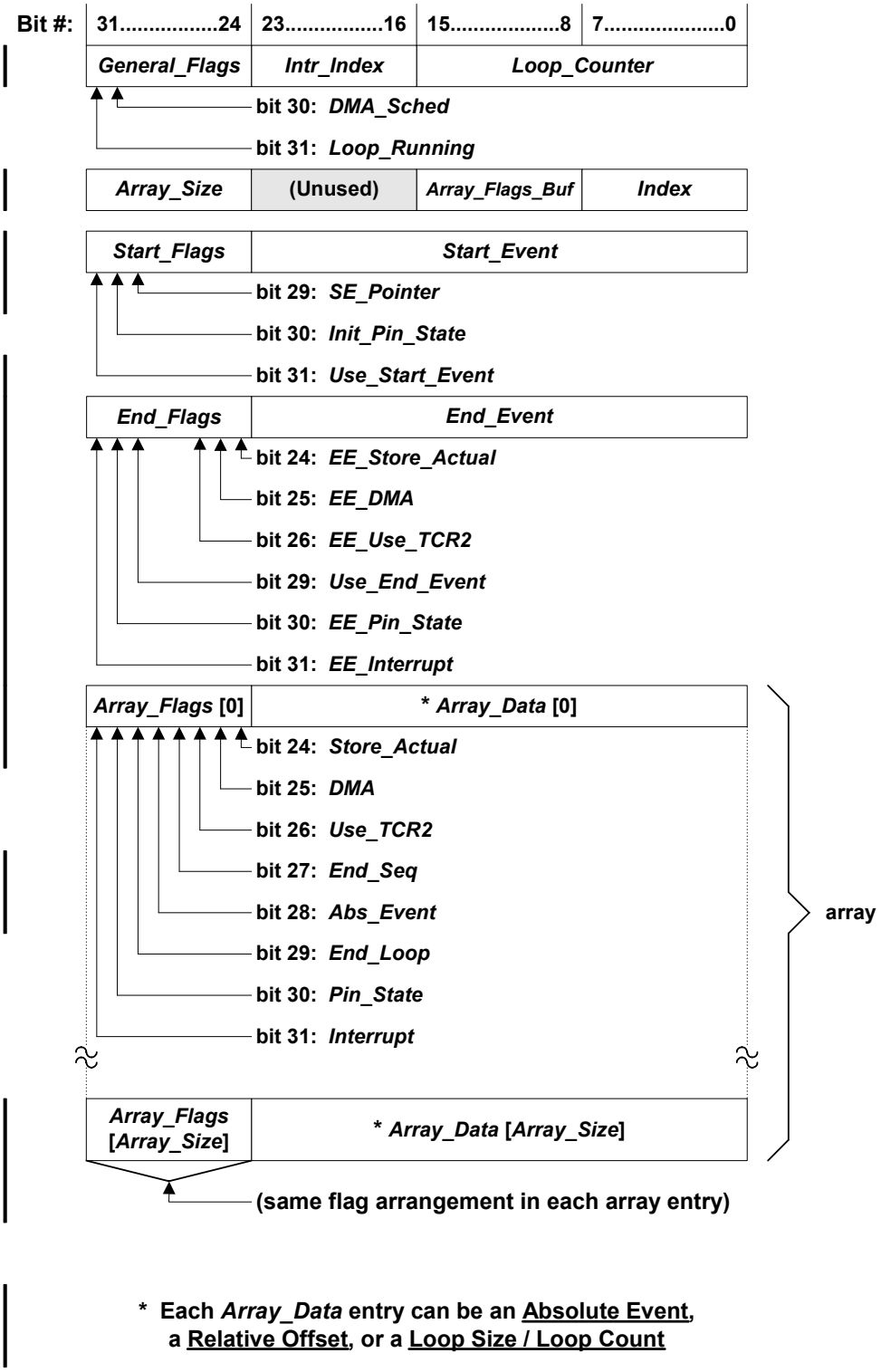


Figure 1 – Hardware Configuration

3. Memory Map

The memory map for the RQOME function is shown in **Figure 2** and **Figure 3**.



* Each *Array_Data* entry can be an Absolute Event, a Relative Offset, or a Loop Size / Loop Count

Figure 2 – Parameter RAM Map

Host Service Requests:

HSR7: Shutdown
HSR6: Request
HSR5: Initialize
HSR4: Update
HSR3: (Unused)
HSR2: (Unused)
HSR1: (Unused)

Entry Point Flags:

Flag0: (Unused)
Flag1: (Unused)

***General_Flags*:**

DMA_Sched
0 = Do not schedule DMA trigger
1 = Schedule DMA trigger
Loop_Running
0 = Repeating loop not in progress
1 = Repeating loop in progress

***Array_Flags*:**

Store_Actual
0 = Do not store actual time/angle at normal event
1 = Store actual time/angle at normal event
DMA
0 = Do not issue DMA trigger to CPU at normal event
1 = Issue DMA trigger to CPU at normal event
Use_TCR2
0 = TCR1 time base for normal event
1 = TCR2 time base for normal event
End_Seq
0 = Entry is not end of sequence
1 = Entry is end of sequence
Abs_Event
(Note: Only valid when *End_Loop* = 0)
0 = Entry is a Relative Offset
1 = Entry is an Absolute Event
End_Loop
0 = Entry is not a Loop Size / Loop Count
1 = Entry is a Loop Size / Loop Count
Pin_State
0 = Set pin low at normal event
1 = Set pin high at normal event
Interrupt
0 = Do not interrupt CPU at normal event
1 = Interrupt CPU at normal event

Function Mode Bits:

FM0: *Event_TCR1_Store_TCR2*
(Note: Only valid when *Use_TCR2* = 0 and *Store_Actual* = 1)
0 = For TCR1 events, store actual event time/angle in TCR1 units
1 = For TCR1 events, store actual event time/angle in TCR2 units
FM1: *Event_TCR2_Store_TCR2*
(Note: Only valid when *Use_TCR2* = 1 and *Store_Actual* = 1)
0 = For TCR2 events, store actual event time/angle in TCR1 units
1 = For TCR2 events, store actual event time/angle in TCR2 units

***Start_Flags*:**

SE_Pointer
0 = *Start_Event* value is not a pointer
1 = *Start_Event* value is a pointer
Init_Pin_State
0 = Set pin low at Initialize HSR
1 = Set pin high at Initialize HSR
Use_Start_Event
0 = Start sequence at last recorded event
1 = Start sequence at *Start_Event*

***End_Flags*:**

EE_Store_Actual
0 = Do not store actual time/angle at *End_Event*
1 = Store actual time/angle at *End_Event*
EE_DMA
0 = Do not issue DMA trigger to CPU at *End_Event*
1 = Issue DMA trigger to CPU at *End_Event*
EE_Use_TCR2
0 = TCR1 time base for *End_Event*
1 = TCR2 time base for *End_Event*
Use_End_Event
0 = Ignore *End_Event*
1 = End sequence at *End_Event*
EE_Pin_State
0 = Set pin low at *End_Event*
1 = Set pin high at *End_Event*
EE_Interrupt
0 = Do not interrupt CPU at *End_Event*
1 = Interrupt CPU at *End_Event*

Figure 3 - Bit Definitions

3.1 RAM Definition

The parameter RAM definitions are summarized in **Table 1**.

Parameter	Range	Units	CPU Access	eTPU Access
General_Flags:				
- *Loop_Running*	0 – 1	Flag	R	R/W
- *DMA_Sched*	0 – 1	Flag	R	R/W
Intr_Index	0 – 0xFF	Array Index #	R	R/W
Loop_Counter	0 – 0xFFFE	Counts	R	R/W
Array_Size	0 – 0xFF	Array Entries - 1	R/W	R
Array_Flags_Buf	0 – 0xFF	Flags	R	R/W
Index	0 – 0xFF	Array Index #	R	R/W
Start_Flags:				
- *Use_Start_Event*	0 – 1	Flag	R/W	R
- *Init_Pin_State*	0 – 1	Flag	R/W	R
- *SE_Pointer*	0 – 1	Flag	R/W	R
Start_Event	0 – 0xFFFFFF	TCRx Units	R/W	R
End_Flags:				
- *EE_Interrupt*	0 – 1	Flag	R/W	R
- *EE_Pin_State*	0 – 1	Flag	R/W	R
- *Use_End_Event*	0 – 1	Flag	R/W	R
- *EE_Use_TCR2*	0 – 1	Flag	R/W	R
- *EE_DMA*	0 – 1	Flag	R/W	R
- *EE_Store_Actual*	0 – 1	Flag	R/W	R
End_Event	0 – 0xFFFFFF	TCRx Units	R/W	R/W
Array_Flags [*Index*] :				
- *Interrupt* [*Index*]	0 – 1	Flag	R/W	R
- *Pin_State* [*Index*]	0 – 1	Flag	R/W	R
- *End_Loop* [*Index*]	0 – 1	Flag	R/W	R
- *Abs_Event* [*Index*]	0 – 1	Flag	R/W	R

- ***End_Seq* [*Index*]**	0 – 1	Flag	R/W	R
- ***Use_TCR2* [*Index*]**	0 – 1	Flag	R/W	R
- ***DMA* [*Index*]**	0 – 1	Flag	R/W	R
- ***Store_Actual* [*Index*]**	0 – 1	Flag	R/W	R
***Array_Data* [*Index*]**				
- **Absolute Event**	0 – 0xFFFFFF	TCRx Units	R/W	R/W
- **Relative Offset**	0 – 0xFFFFFF	TCRx Units	R/W	R/W
- **Loop Size /**	1 – 0xFF /	Loop Entries /	R/W	R/W
Loop Count	1 – 0xFFFE	Counts	R/W	R/W
Function Mode Bits:				
- Event_TCR1_Store_TCR2	0 – 1	Flag	R/W	R
- Event_TCR2_Store_TCR2	0 – 1	Flag	R/W	R

Table 1 - RAM Definitions

3.1.1 CPU Write / eTPU Read

The following parameters are written by the CPU and read by the eTPU:

Array_Size **Bits 31-24 –** The total number of entries in the data array minus one. ***Array_Size*** is used to limit the value of ***Index*** and determine when the last ***Array_Data*** entry has been reached.

NOTE: Care must be taken when writing to *Array_Size* since it shares a 32-bit word with *Index* and *Array_Flags_Buf* (both of which are written by the eTPU). *Array_Size* should only be written with an 8-bit write operation that does not affect the other parameters.

Start_Flags **Bits 31-29:**

Use_Start_Event **Bit 31 –** Set (1) if ***Start_Event*** (or the value it points to) is to be used to time the beginning of an event sequence. Cleared (0) if the last event of a previous

sequence is to be used to time the beginning of an event sequence.

Init_Pin_State **Bit 30 –** The desired value of the output pin state at initialization. ***Init_Pin_State*** must be written before an **Initialize HSR** is issued to the eTPU.

SE_Pointer **Bit 29 –** Set (1) if ***Start_Event*** contains the byte address of another eTPU RAM location holding the desired start time/angle value. (This byte address assumes that the eTPU RAM space begins at address 0.) Cleared (0) if ***Start_Event*** contains the desired start time/angle value itself.

Start_Event **Bits 23-0** - If ***Use_Start_Event*** = 1 and ***SE_Pointer*** = 0, ***Start_Event*** is an absolute time/angle event from which ***Array_Data*** **[0]** will be timed.

If ***Use_Start_Event*** = 1 and ***SE_Pointer*** = 1, ***Start_Event*** is the byte address of another eTPU RAM location containing an absolute time/angle event from which ***Array_Data*** **[0]** will be timed. (This byte address assumes that the eTPU RAM space begins at address 0.)

Care must be taken that ***Start_Event*** (or the value it points to) + ***Array_Data*** **[0]** is not more than 0x800000 timer counts in the future. ***Start_Event*** (and/or the value it points to) must be written **before** a **Request HSR** is issued to the eTPU. ***Start_Event*** **(or the value it points to) is always an Absolute Event and uses the same time base as the first array entry (*Array_Data* [0]).**

If ***Use_Start_Event*** = 0, ***Start_Event*** is ignored and the last event of a previous sequence is used in its place.

NOTE: The user must ensure that *Start_Event* (and/or the value it points to) contains valid data whenever *Use_Start_Event* = 1.

End_Flags	**Bits 31-29, 26-24:**
EE_Interrupt	**Bit 31 –** Set (1) if the eTPU is to issue an interrupt to the host CPU at ***End_Event***; cleared (0) otherwise.
EE_Pin_State	**Bit 30 –** The desired value of the output pin state at ***End_Event***.
Use_End_Event	**Bit 29 –** Set (1) if ***End_Event*** is to be used to terminate an event sequence; cleared (0) if an event sequence is to terminate normally.
EE_Use_TCR2	**Bit 26 –** Set (1) if ***End_Event*** uses TCR2 (time or angle) as its time base; cleared (0) if ***End_Event*** uses TCR1 (time only) as its time base.
EE_DMA	**Bit 25 –** Set (1) if the eTPU is to issue a DMA trigger to the host CPU at ***End_Event***; cleared (0) otherwise. **Note that not all eTPU channels have DMA trigger capability. The user must verify that the desired channel has this capability.**
EE_Store_Actual	**Bit 24 –** Set (1) if the eTPU is to store the actual time or angle of ***End_Event*** back into ***End_Event*** when the event itself occurs (in units determined by ***Event_TCR1_Store_TCR2*** and ***Event_TCR2_Store_TCR2***); cleared (0) otherwise.
Array_Flags **[***Index***]**	**Bits 31-24:**
Interrupt **[***Index***]**	**Bit 31 –** Set (1) if the eTPU is to issue an interrupt to the host CPU at the event defined by the associated ***Array_Data*** entry; cleared (0) otherwise.
Pin_State **[***Index***]**	**Bit 30 –** The desired value of the output pin state at the event defined by the associated ***Array_Data*** entry.
End_Loop **[***Index***]**	**Bit 29 –** Set (1) if the associated ***Array_Data*** entry is a **Loop Size / Loop Count**; cleared (0) otherwise. When ***End_Loop*** = 0, ***Abs_Event*** will determine the status of the associated ***Array_Data*** entry.
Abs_Event **[***Index***]**	**Bit 28 –** Set (1) if the associated ***Array_Data*** entry is an **Absolute Event**; cleared (0) if the associated ***Array_Data*** entry is a **Relative Offset**. If the

associated *Array_Data* entry is a **Loop Size / Loop Count** (ie, *End_Loop* = 1), *Abs_Event* will be ignored.

End_Seq **[***Index***]** **Bit 27 –** Set (1) if the associated *Array_Data* entry is the final event in the sequence; cleared (0) otherwise. **Note that if no *End_Seq* [*Index*] bit is set, the event sequence will repeat indefinitely.**

NOTE: If *End_Seq* = 1 for a Loop Size / Loop Count entry (ie, *End_Loop* = 1), it indicates that the sequence will end when the loop has finished repeating. Thus, if Loop Count = 0xFFFF (repeat loop indefinitely), setting *End_Seq* = 1 will have no effect.

Use_TCR2 **[***Index***]** **Bit 26 –** Set (1) if the associated *Array_Data* entry uses TCR2 (time or angle) as its time base; cleared (0) if the associated *Array_Data* entry uses TCR1 (time only) as its time base.

DMA **[***Index***]** **Bit 25 –** Set (1) if the eTPU is to issue a DMA trigger to the host CPU at the event defined by the associated *Array_Data* entry; cleared (0) otherwise. **Note that not all eTPU channels have DMA trigger capability. The user must verify that the desired channel has this capability.**

Store_Actual **[***Index***]** **Bit 24 –** Set (1) if the eTPU is to store the actual time or angle of the event back into the associated *Array_Data* entry when the event occurs (in units determined by ***Event_TCR1_Store_TCR2*** and ***Event_TCR2_Store_TCR2***); cleared (0) otherwise. **Note that extreme care must be exercised when using this feature in a repeating loop!**

3.1.1.1 Function Mode Bits

The two Function Mode (FM) bits are located in the ETPUCxSCR register. They are written by the CPU and read by the eTPU:

Event_TCR1_Store_TCR2

FM0 - Set (1) if, **for TCR1 events**, the actual event time/angle is to be stored in TCR2 units; cleared (0) if,

for TCR1 events, the actual event time is to be stored in TCR1 units. **Note that *Event_TCR1_Store_TCR2* applies both to normal events and to *End_Event*.** ***Event_TCR1_Store_TCR2*** will be read by the eTPU only when ***Use_TCR2* [*Index*]** (or ***EE_Use_TCR2***) = 0 and ***Store_Actual* [*Index*]** (or ***EE_Store_Actual***) = 1.

Event_TCR2_Store_TCR2

FM1 - Set (1) if, **for TCR2 events**, the actual event time/angle is to be stored in TCR2 units; cleared (0) if, **for TCR2 events**, the actual event time is to be stored in TCR1 units. **Note that *Event_TCR2_Store_TCR2* applies both to normal events and to *End_Event*.** ***Event_TCR2_Store_TCR2*** will be read by the eTPU only when ***Use_TCR2* [*Index*]** (or ***EE_Use_TCR2***) = 1 and ***Store_Actual* [*Index*]** (or ***EE_Store_Actual***) = 1.

3.1.2 eTPU Write / CPU Read

The following parameters are written by the eTPU and read by the CPU:

General_Flags **Bits 31-30:**

Loop_Running **Bit 31 –** Set (1) when a **Loop Size / Loop Count** entry is first encountered in ***Array_Data*** (start of repeating loop), unless the value of **Loop Count** is 0xFFFF (when **Loop Count** = 0xFFFF, ***Loop_Running*** is never set). Cleared (0) when ***Loop_Counter*** is decremented to 0 (end of repeating loop) or ***End_Event*** is reached. Also cleared (0) when either an **Initialize HSR** or a **Request HSR** is issued by the CPU, or when a link is received from another eTPU channel.

DMA_Sched **Bit 30 –** Set (1) to indicate if the eTPU is to issue a DMA trigger to the host CPU; cleared (0) at the start of an output event service routine, when a link is received from another eTPU channel, and when an **Initialize**, **Request** or **Update HSR** is issued by the CPU.

Intr_Index **Bits 23-16** - The number of the ***Array_Data*** event indicated by ***Index*** at which the eTPU last issued an interrupt to the host CPU.

NOTE: If the interrupt was caused by *End_Event*, then *Intr_Index* will contain 0xFF.

Loop_Counter **Bits 15-0** - Down-counter used to implement repeating loops within array. When a **Loop Size / Loop Count** entry is encountered (ie, ***End_Loop*** = 1), ***Loop_Counter*** is initialized to the **Loop Count** value (bits 15-0) and is decremented each succeeding time through the loop. When ***Loop_Counter*** = 0, the loop is finished.

NOTE: If the value of Loop Count is 0xFFFF, *Loop_Counter* will be re-initialized to this value at each output event, causing the loop to repeat indefinitely.

Array_Flags_Buf **Bits 15-8 –** Buffered copy of ***Array_Flags*** **[*Index*]** (when a normal event is set up) or ***End_Flags*** (when ***End_Event*** occurs).

Index **Bits 7-0** - Index into ***Array_Data*** indicating the event currently being accessed. Minimum value = 0, maximum value = ***Array_Size***.

3.1.3 eTPU Write / CPU Write

The following parameters are written by the eTPU and by the CPU:

End_Event **Bits 23-0** - If ***Use_End_Event*** = 1, ***End_Event*** is an absolute time/angle event at which a sequence in progress will be terminated. The event will have characteristics determined by ***End_Flags***. ***End_Event*** **is always an Absolute Event.**

If ***End_Event*** occurs and ***EE_Store_Actual*** = 1, the actual time or angle of the event will be stored back into ***End_Event*** by the eTPU (in units determined by

Event_TCR1_Store_TCR2 and *Event_TCR2_Store_TCR2*).

If ***Use_End_Event*** = 0, ***End_Event*** and all its associated bits will be ignored.

NOTE: The user must ensure that *End_Event* contains valid data whenever *Use_End_Event* = 1.

Array_Data **[***Index***]** **Bits 23-0** - An *Array_Data* entry will be interpreted as an **Absolute Event** , as a **Relative Offset**, or as a **Loop Size / Loop Count** depending on the state of the associated flags ***End_Loop*** and ***Abs_Event***.

Array_Data will be interpreted as an **Absolute Event** (time/angle) when the associated ***End_Loop*** = 0 and the associated ***Abs_Event*** = 1 (see **Figure 4**). In this case the **Absolute Event** entry is the value of TCRx at which the current event is scheduled to occur.

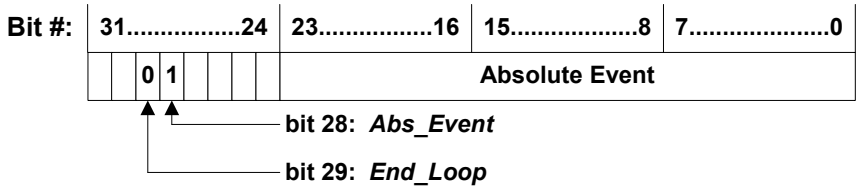


Figure 4 – *Array_Data*: Absolute Event

Array_Data will be interpreted as a **Relative Offset** (time/angle) when the associated ***End_Loop*** = 0 and the associated ***Abs_Event*** = 0 (see **Figure 5**). In this case the **Relative Offset** entry is the number of TCRx counts after the previous event at which the current event is scheduled to occur. **Note that the actual time/angle of the previous event is used to time Relative Offsets rather than the written (desired) time/angle. These values will be identical unless an event is already in the past when the eTPU services it. (See Section 4.6.)**

The 1st entry – *Array_Data* [0] – must be a Relative Offset!

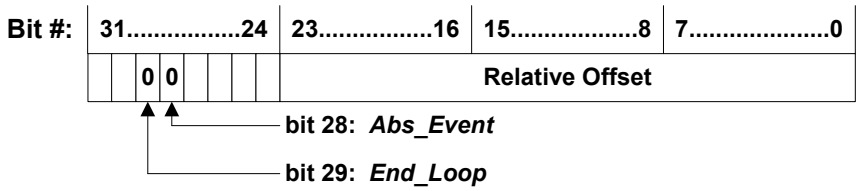


Figure 5 - *Array_Data*: Relative Offset

Array_Data will be interpreted as a **Loop Size / Loop Count** when the associated ***End_Loop*** = 1. In this case **bits 23-16 contain Loop Size** and **bits 15-0 contain Loop Count** (see **Figure 6**). **Loop Size** is the total number of array entries comprising the loop (not including the **Loop Size / Loop Count** entry itself). **Loop Count** is the number of times the loop within the event sequence is to be repeated. The **Loop Size / Loop Count** entry must be located immediately **after** the last loop entry.

NOTE: The minimum legitimate value of both Loop Size and Loop Count is 1. If either Loop Size or Loop Count is set to 0, the eTPU will treat it as though it were set to 1.

NOTE: The normal maximum value of Loop Count is 0xFFFE. If Loop Count is set to 0xFFFF, the eTPU will treat it as a special case that will cause the loop to repeat indefinitely!

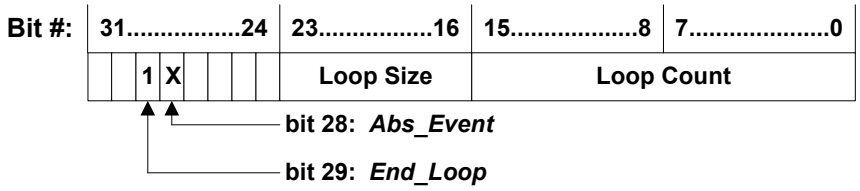


Figure 6 – *Array_Data*: Loop Size / Loop Count

When a normal event occurs and the associated ***Store_Actual*** = 1, the actual time or angle of the event will be stored back into ***Array_Data*** **[*Index*]** by the eTPU (in units determined by ***Event_TCR1_Store_TCR2*** and ***Event_TCR2_Store_TCR2***).

NOTE: Updating *Array_Data* and/or any associated bits while an event sequence is in progress can cause unexpected results in the output signal!

3.2 Initialization

The CPU should initialize the RQOME function as follows:

1. Write the encoded value for the RQOME primitive to the channel's field within the correct channel function select register (CFSx). See the Application Implementation document.
2. Write desired value to ***Init_Pin_State***.
3. Issue an **Initialize Host Service Request (%101)**.
4. Enable the RQOME channel by assigning to it a non-zero priority (CPR > %00).

When an **Initialize HSR** is issued, the eTPU will respond by performing the following actions:

1. Configure the RQOME channel to Either Match (Blocking) mode. Using this mode, a normal match event and ***End_Event*** (if selected) may be set up simultaneously with independent time bases. The first of these two events to occur will block the other event from occurring. (If both events occur at the same time, ***End_Event*** takes priority.) In addition, both of the

eTPU's time bases (TCR1 and TCR2) will be captured on every output event that occurs.

Normal match events are set up by writing to the internal eTPU register ERTB. ***End_Event*** (if selected) is set up by writing to the internal eTPU register ERTA. Each type of event may be set up to use either of the eTPU's time bases (TCR1 and TCR2). When any output event (of whatever type) occurs, ERTA will contain the captured event in TCR1 units and ERTB will contain the captured event in TCR2 units.

2. Set/clear the output pin per ***Init_Pin_State***.
3. Set ***Loop_Running*** = 0 and ***DMA_Sched*** = 0.

4. Operation

4.1 Examples

The following two examples are meant to illustrate the basic functionality of the RQOME algorithm.

The first example (see **Figure 7**) shows how ***Init_Pin_State*** and ***Start_Event*** are used at the beginning of an event sequence (***End_Event*** is disabled). The example further shows how both **Absolute Events** and **Relative Offsets** may be used in the same event sequence, how a repeating loop operates, and how an interrupt and a DMA trigger are requested at a given event by the CPU. Finally, since none of the ***End_Seq*** bits are set, the example shows the entire sequence re-starting itself when the last event is reached.

The second example (see **Figure 8**) illustrates a repeating loop terminated prematurely by ***End_Event***. In addition, the two **Relative Offsets** in the repeating loop use different time bases.

Note that the ***Store_Actual*** bits for looping elements are never set.

NOTE: Extreme care must be taken whenever no *End_Seq* bit = 1 (ie, the sequence is set to repeat indefinitely) and one or both of the following conditions exist: (1) any *Abs_Event* bit = 1; (2) any *Store_Actual* bit = 1.

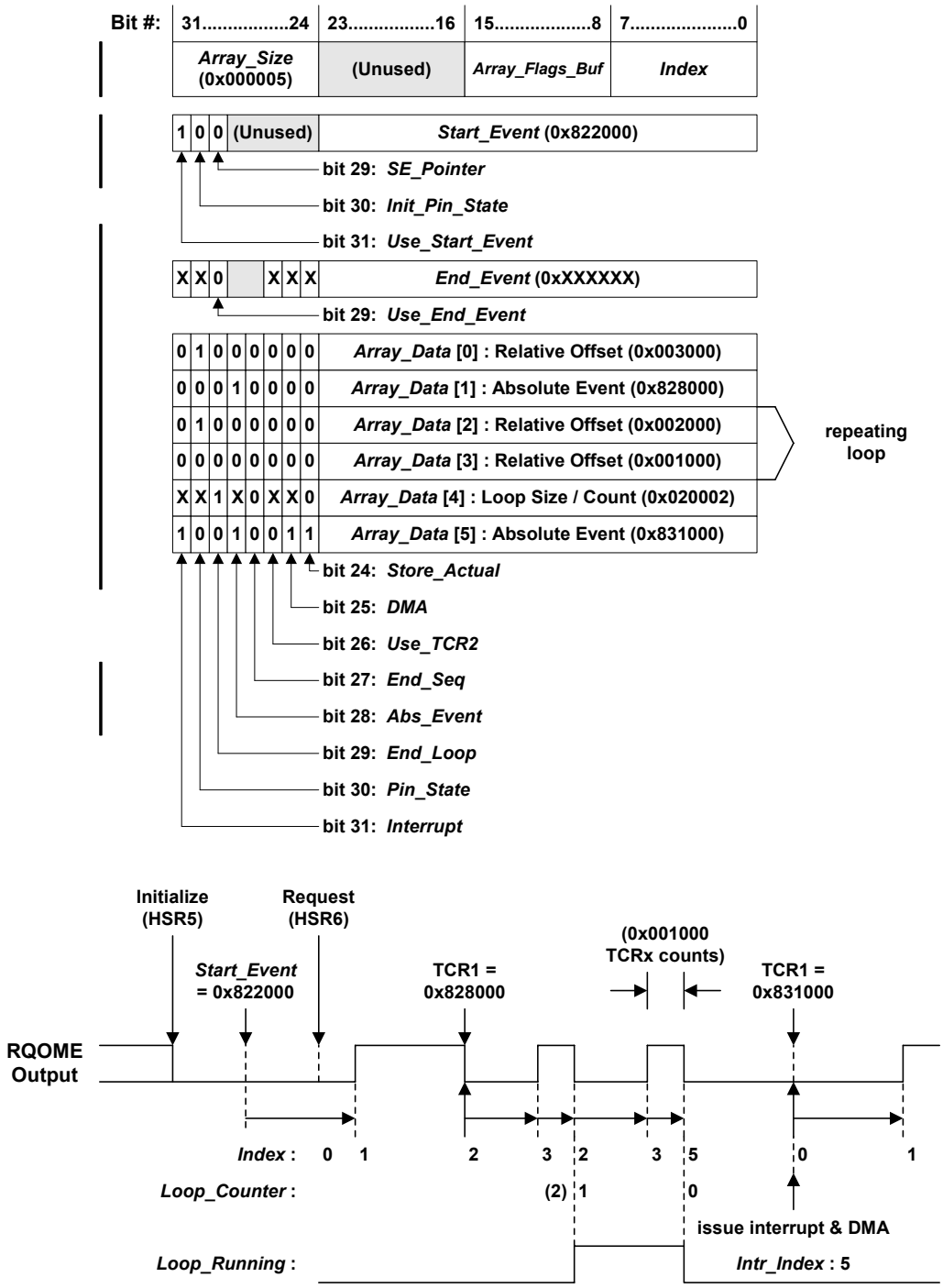


Figure 7 - Event Sequence with Indefinitely Repeating Loop

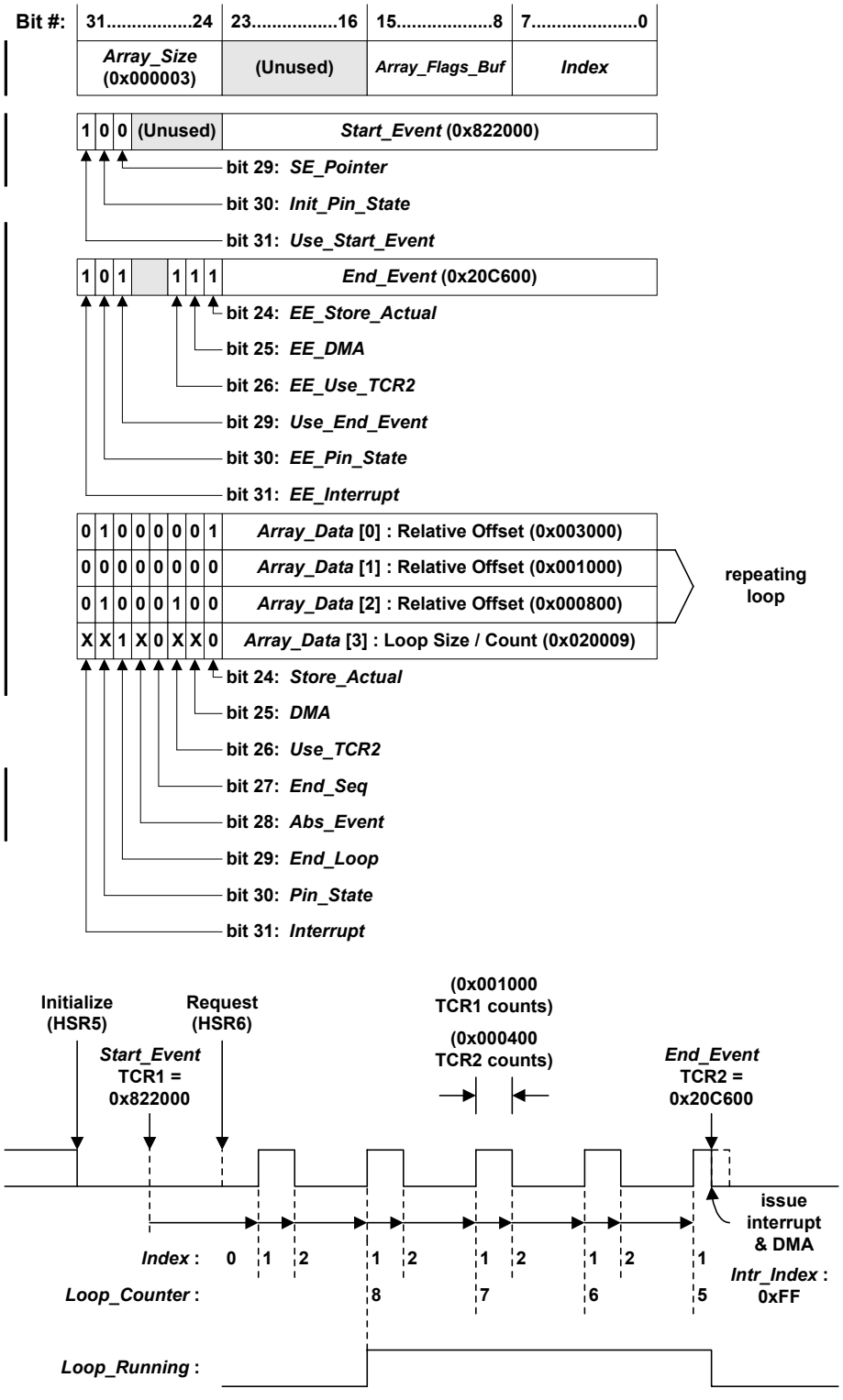


Figure 8 - Event Sequence Prematurely Terminated by *End_Event*

4.2 Requesting an Event Sequence

After initialization, the CPU may initiate an event sequence by performing the following actions:

1. Write the desired number of array entries minus one to ***Array_Size***. (For example, if there are 10 entries in the array, set ***Array_Size*** = 9.) **Note that *Array_Size* should always be written with an 8-bit write operation to avoid a potential conflict with *Index* and *Array_Flags_Buf*!**

2. Write desired values to ***Start_Event***, ***Use_Start_Event*** and ***SE_Pointer***. **If *SE_Pointer* = 0, *Start_Event* is an absolute time/angle. If *SE_Pointer* = 1, *Start_Event* is the byte address of another eTPU RAM location containing an absolute time/angle. (This byte address assumes that the eTPU RAM space begins at address 0.) In either case, the absolute time/angle uses the same time base as the <u>first</u> array entry (*Array_Data* [0]). The user must ensure that *Start_Event* contains valid data whenever *Use_Start_Event* = 1.**

3. Write desired values to ***End_Event*** and ***Use_End_Event***. ***End_Event*** **is an absolute time/angle that has characteristics defined by *End_Flags*. The user must ensure that *End_Event* contains valid data whenever *Use_End_Event* = 1.**

4. Write desired values to each ***Array_Data*** entry that is to be utilized and to the ***Array_Flags*** associated with each entry. Each ***Array_Data*** entry may be configured as an **Absolute Event**, a **Relative Offset**, or a **Loop Size / Loop Count**. **The first event in the sequence – *Array_Data* [0] - <u>must</u> be a Relative Offset.**

5. If the ***Store_Actual*** bit for any array entry has been set (1), make sure that the bits ***Event_TCR1_Store_TCR2*** and ***Event_TCR2_Store_TCR2*** are set to the desired states.

6. Issue a **Request Host Service Request (%110)** to initiate the event sequence.

When a **Request HSR** is issued, the eTPU will respond by performing the following actions:

1. Cancel any pending match.

2. Set ***Loop_Running*** = 0 and ***DMA_Sched*** = 0.

3. Initialize ***Index*** to 0 to point to the first entry in the array, ie, ***Array_Data*** **[0]**.

4. If ***Use_Start_Event*** = 0 and ***Use_TCR2*** **[0]** = 0, set up the first event as follows:

   ERTB = ERTA (last event in TCR1 units) + ***Array_Data*** **[0]**

   If ***Use_Start_Event*** = 0 and ***Use_TCR2*** **[0]** = 1, set up the first event as follows:

   ERTB = ERTB (last event in TCR2 units) + ***Array_Data*** **[0]**

   If ***Use_Start_Event*** = 1 and ***SE_Pointer*** = 0, set up the first event as follows:

   ERTB = ***Start_Event*** + ***Array_Data*** **[0]**

   If ***Use_Start_Event*** = 1 and ***SE_Pointer*** = 1, set up the first event as follows:

   ERTB = **[*Start_Event*]** + ***Array_Data*** **[0]**

   ERTB is <u>always</u> set up to capture events in TCR2 units.

   The time base and the pin state of the first event will be determined by ***Use_TCR2*** **[0]** and ***Pin_State*** **[0]**, respectively.

5. Store ***Array_Flags*** for first event in ***Array_Flags_Buf***.

6. If ***Use_End_Event*** = 1, set up the end event as follows:

   ERTA = ***End_Event***

   ERTA is <u>always</u> set up to capture events in TCR1 units.

   The time base and the pin state of ***End_Event*** will be determined by ***EE_Use_TCR2*** and ***EE_Pin_State***, respectively.

<u>WARNING:</u> A match event can be timed up to 0x800000 timer counts in the future. An event that is farther in the future than this will be treated by the eTPU as a **past** event, and generated immediately.

4.3 Normal Match Event

When a normal output match event (ie, not ***End_Event***) occurs, the eTPU will respond by performing the following actions:

1. Cancel any pending normal match.
2. If ***Store_Actual*** (in ***Array_Flags_Buf***) = 1, store the actual event time/angle in ***Array_Data*** **[*Index*]** as described below.

   If ***Use_TCR2*** (in ***Array_Flags_Buf***) = 0 and ***Event_TCR1_Store_TCR2*** = 0, **OR** ***Use_TCR2*** (in ***Array_Flags_Buf***) = 1 and ***Event_TCR2_Store_TCR2*** = 0:

   Array_Data **[*Index*]** = ERTA (actual event in TCR1 counts)

   If ***Use_TCR2*** (in ***Array_Flags_Buf***) = 0 and ***Event_TCR1_Store_TCR2*** = 1, **OR** ***Use_TCR2*** (in ***Array_Flags_Buf***) = 1 and ***Event_TCR2_Store_TCR2*** = 1:

   Array_Data **[*Index*]** = ERTB (actual event in TCR2 counts)
3. If ***Interrupt*** (in ***Array_Flags_Buf***) = 1, write the number of ***Index*** to ***Intr_Index*** and issue an interrupt to the host CPU.
4. If ***DMA*** (in ***Array_Flags_Buf***) = 1, set ***DMA_Sched*** = 1; otherwise, set ***DMA_Sched*** = 0.
5. If ***End_Seq*** (in ***Array_Flags_Buf***) = 1, set ***Loop_Running*** = 0 and jump to **Step 16**.
6. If ***Index*** >= ***Array_Size***, then reset ***Index*** to the first entry in the array (***Array_Data*** **[0]**) and jump to **Step 12**. Otherwise, increment ***Index*** to the next ***Array_Data*** entry and continue.
7. If the next ***Array_Data*** entry is an **Absolute Event** or a **Relative Offset** (***End_Loop*** **[*Index*]** = 0), jump to **Step 13**. Otherwise, the next ***Array_Data*** entry must be a **Loop Size / Loop Count** (***End_Loop*** **[*Index*]** = 1), so continue.
8. If no loop is currently in progress (***Loop_Running*** = 0), initialize ***Loop_Counter*** with the **Loop Count** value contained in ***Array_Data*** **[*Index*]**; also, if **Loop Count** < 0xFFFF, set ***Loop_Running*** = 1.
9. Decrement ***Loop_Counter***. If the repeating loop is not complete (***Loop_Counter*** > 0), set ***Index*** to point to the start of the repeating loop by

subtracting the **Loop Size** value contained in ***Array_Data* [*Index*]** (limiting the result to 0) and jump to **Step 13**. Otherwise, continue.

10. If ***End_Seq* [*Index*]** = 1, set ***Loop_Running*** = 0 and jump to **Step 16**.
11. If ***Index*** >= ***Array_Size***, reset ***Index*** to the first entry in the array (***Array_Data* [0]**); otherwise, increment ***Index*** to the next ***Array_Data*** entry.
12. Set ***Loop_Running*** = 0.
13. If the next event is an **Absolute Event** (***Abs_Event* [*Index*]** = 1), set it up as follows:

    ERTB = ***Array_Data* [*Index*]**

    If the next event is a **Relative Offset** (***Abs_Event* [*Index*]** = 0) and ***Use_TCR2* [*Index*]** = 0, set it up as follows:

    ERTB = ERTA (last event in TCR1 counts) + ***Array_Data* [*Index*]**

    If the next event is a **Relative Offset** (***Abs_Event* [*Index*]** = 0) and ***Use_TCR2* [*Index*]** = 1, set it up as follows:

    ERTB = ERTB (last event in TCR2 counts) + ***Array_Data* [*Index*]**

    ERTB is always set up to capture events in TCR2 units.

    The time base and the pin state of the next event will be determined by ***Use_TCR2* [*Index*]** and ***Pin_State* [*Index*]**, respectively.
14. Store ***Array_Flags*** for next event in ***Array_Flags_Buf***.
15. If ***Use_End_Event*** = 1, set up the end event as follows:

    ERTA = ***End_Event***

    ERTA is always set up to capture events in TCR1 units.

    The time base and the pin state of the end event will be determined by ***EE_Use_TCR2*** and ***EE_Pin_State***, respectively.
16. If ***DMA_Sched*** = 1, issue a DMA trigger to the host CPU.

4.4 End Event

When ***End_Event*** occurs, the eTPU will respond by setting ***Index*** = 0xFF, storing ***End_Flags*** in ***Array_Flags_Buf*** (with the bit occupying the position of ***End_Seq*** always set = 1), and performing the same algorithm as for a normal event (see **Section 4.3**).

4.5 Updating End Event

All data contained in the array may be updated by the user at any time. Indeed, if no ***End_Seq*** bit is set, this data must be continuously updated by the user. However, any data contained in the array – new or otherwise - will only take effect on an output match event. **There is no provision in the RQOME function for changes in array data to take immediate effect.**

The user may update ***End_Flags*** and/or ***End_Event*** in exactly the same manner as array data, if desired. However, the user also has the option to have changes to ***End_Flags*** and/or ***End_Event*** applied immediately to an event sequence in progress. To achieve this, the CPU must perform the following actions:

1. Write desired values to ***End_Event*** and ***End_Flags***. **The user must ensure that *End_Event* contains valid data whenever *Use_End_Event* = 1.**
2. Issue an **Update Host Service Request (%100)**

When an **Update HSR** is issued, the eTPU will respond by performing the following actions:

1. Set ***DMA_Sched*** = 0.
2. If ***Use_End_Event*** = 1, set up the end event as follows:

   ERTA = ***End_Event***

   ERTA is always set up to capture events in TCR1 units.

   The time base and the pin state of the end event will be determined by ***EE_Use_TCR2*** and ***EE_Pin_State***, respectively.

4.6 Timing Past Events

The RQOME algorithm uses a "greater than or equal to" comparator to time output events. This means that if an event is set up that is up to 0x7FFFFF timer counts in the past, the eTPU will generate the event immediately. (Events that are

more than 0x7FFFFF timer counts in the past will be treated as future events.) Furthermore, when computing an event using a **Relative Offset**, the RQOME algorithm uses the **actual time/angle** of the event just past, rather than the desired time/angle that was written by the user. Normally, these two values will be the same. However, they will be different if a past event was set up by the eTPU, whether this was due to the user's preference or to system latencies. The user must be aware that such an occurrence can affect the timing of any future events that are based upon it.

4.7 Array Flag Buffering

Whenever the RQOME primitive sets up a normal output event, ***Array_Flags*** for that pending event is stored in ***Array_Flags_Buf***. When the actual event occurs, the primitive tests the flags contained in ***Array_Flags_Buf*** to determine whether or not to issue an interrupt and/or a DMA trigger to the host, whether or not to store the actual time/angle of the event back into ***Array_Data*** **[*Index*]**, and whether or not the final event in the sequence has been reached. The reason for this buffering is so that, once an event has been set up by the eTPU, the host may re-load these flags immediately with new data if desired.

When ***End_Event*** occurs, ***End_Flags*** is stored in ***Array_Flags_Buf***.

4.8 Global Exception

Under certain error conditions, the RQOME primitive will issue a global exception to the host and write its channel number to the global variable ***Cause_Of_Exception***. Below are the conditions that cause RQOME to generate a global exception:

1. An HSR is issued to the RQOME primitive that has not been defined.

4.9 Links From Other eTPU Channels

If the RQOME primitive receives a link from another eTPU channel, it will respond by executing the same algorithm as for a **Request HSR** (see **Section** 4.2).

4.10 Aborting an Event Sequence

An event sequence in progress can be aborted by issuing an **Initialize HSR** (see **Section 3.2**). The **Initialize HSR** will cancel any pending match and prepare the

RQOME channel for a **Request HSR**. The user must always set the ***Init_Pin_State*** bit to the desired state prior to issuing an **Initialize HSR**.

Due to the inherent latency of the eTPU in responding to events, it is possible for the eTPU to issue an interrupt and/or DMA trigger to the CPU from a pending match slightly after an Initialize HSR has been issued to abort an event sequence in progress.

4.11 Shutting Down the RQOME Function

This function can be disabled by issuing a **Shutdown Host Service Request (%111)**, waiting for the HSR bits to clear, and setting the priority level to disabled (%00).

4.12 Switching from RQOME to Another eTPU Function

To switch from the RQOME function to another eTPU function, the CPU should shut down the RQOME function (see **Section 4.11**) and then follow the directions for initializing the new function.

4.13 Function Latency

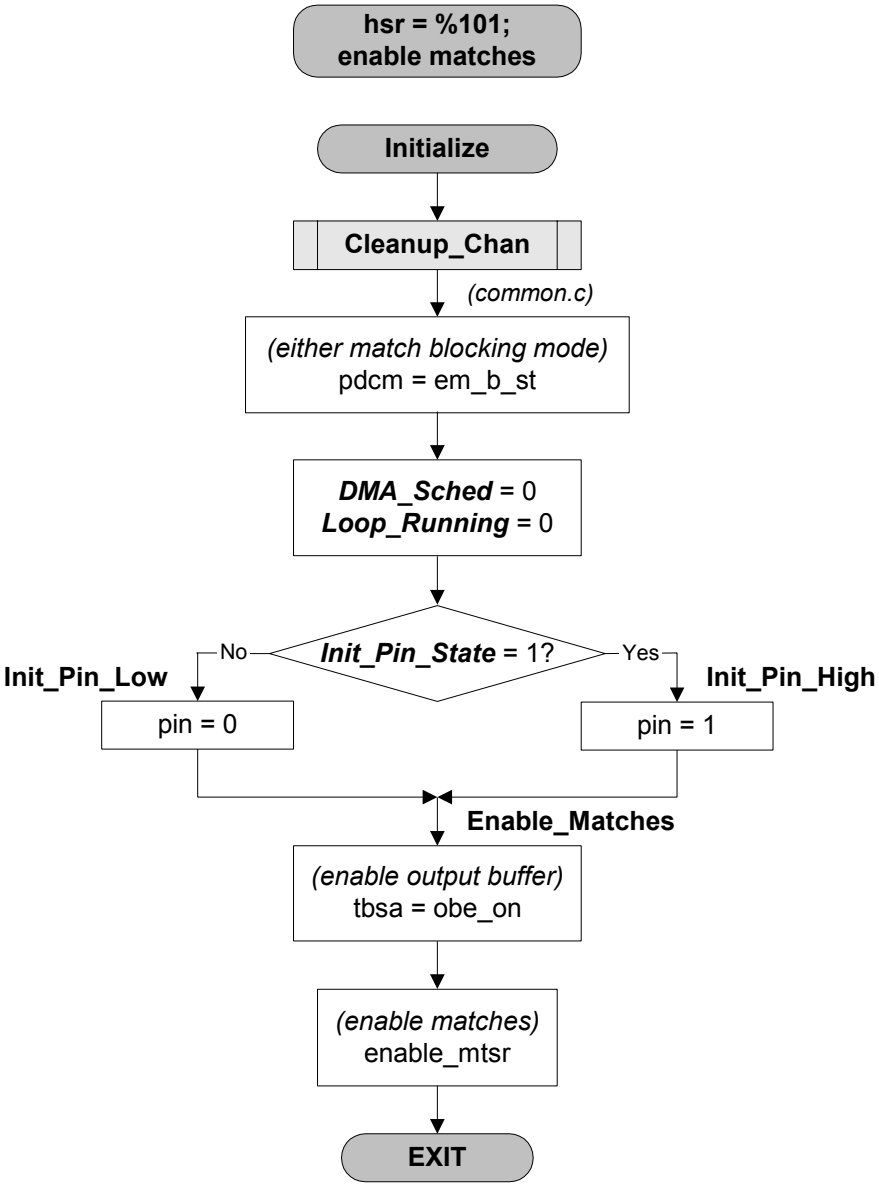
The worst-case execution times for the various threads of the RQOME function are shown below (see the Application Implementation document to convert microcycles into real time based on crystal frequency). Note that these values do not take into account latencies due to other functions in the application, priorities, etc.

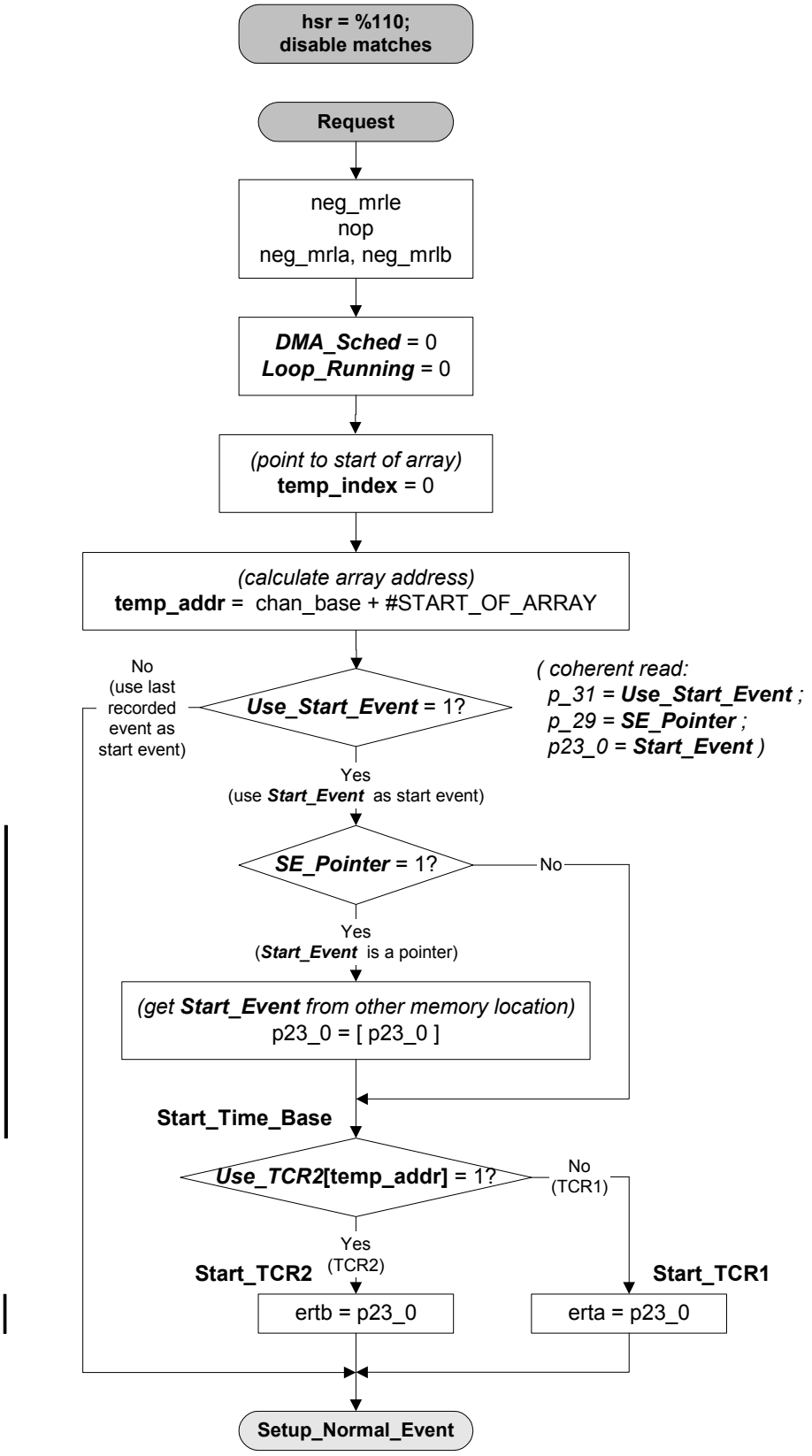
RQOME Thread	Worst-Case µcycles *	RAM Accesses
Shutdown HSR	**8**	**0**
Request HSR (no *End_Event*)	**37**	**9**
Initialize HSR	**15**	**2**
Update HSR	**16**	**5**
Link (no *End_Event*)	**39**	**9**
Output Event:		
Absolute Event (w/o *End_Event*)	**57**	**14**
Absolute Event (w/ *End_Event*)	**63**	**14**
Relative Offset (w/o *End_Event*)	**59**	**15**
Relative Offset (w/ *End_Event*)	**65**	**15**
Loop Count (w/o *End_Event*)	**84**	**21**
Loop Count (w/ *End_Event*)	**90**	**21**
End_Event	**41**	**14**

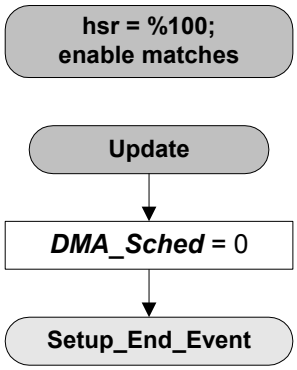
Total number of instructions: 165

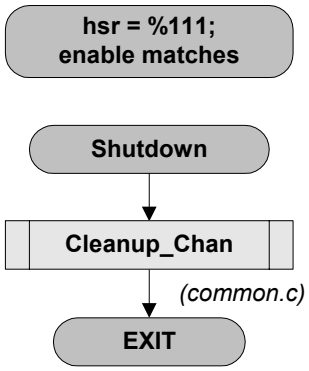
* These numbers do not consider potential collisions while accessing RAM.

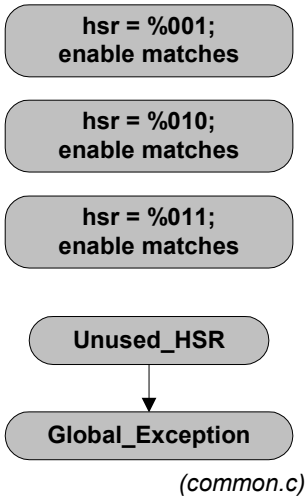
5. Flowcharts

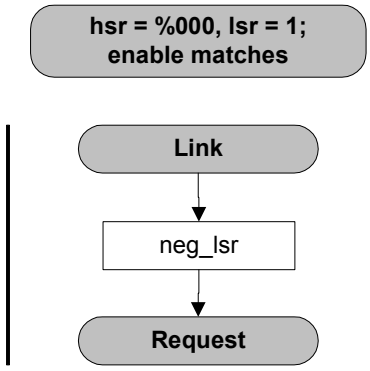


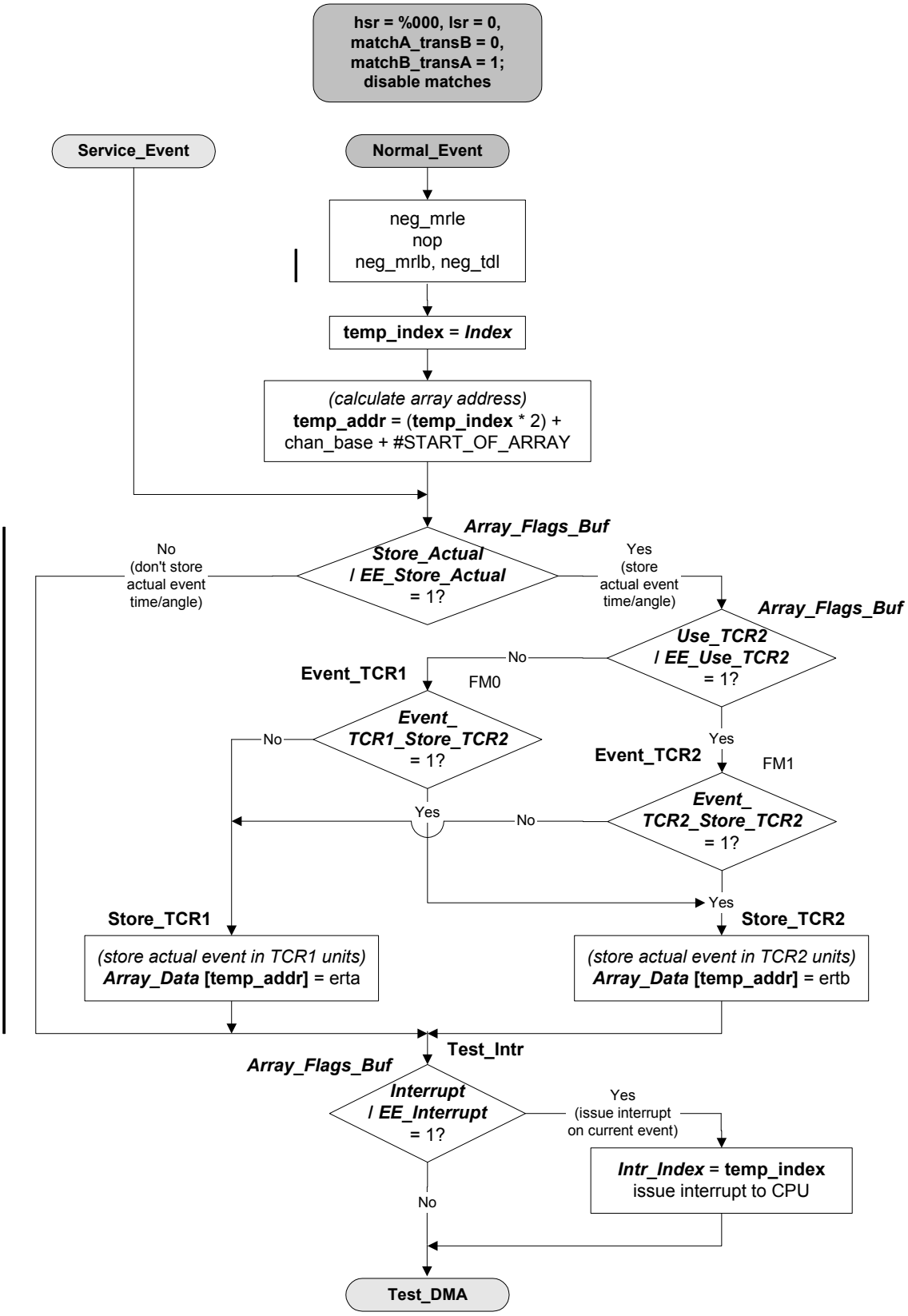

hsr = %110;
disable matches
Request
neg_mrle
nop
neg_mrla, neg_mrlb
DMA_Sched = 0
Loop_Running = 0
(point to start of array)
temp_index = 0
(calculate array address)
temp_addr = chan_base + #START_OF_ARRAY
No
(use last recorded event as start event)
Use_Start_Event = 1?
(coherent read:
p_31 = Use_Start_Event ;
p_29 = SE_Pointer ;
p23_0 = Start_Event)
Yes
(use Start_Event as start event)
SE_Pointer = 1?
No
Yes
(Start_Event is a pointer)
(get Start_Event from other memory location)
p23_0 = [p23_0]
Start_Time_Base
Use_TCR2[temp_addr] = 1?
No
(TCR1)
Yes
(TCR2)
Start_TCR2
Start_TCR1
ertb = p23_0
erta = p23_0
Setup_Normal_Event

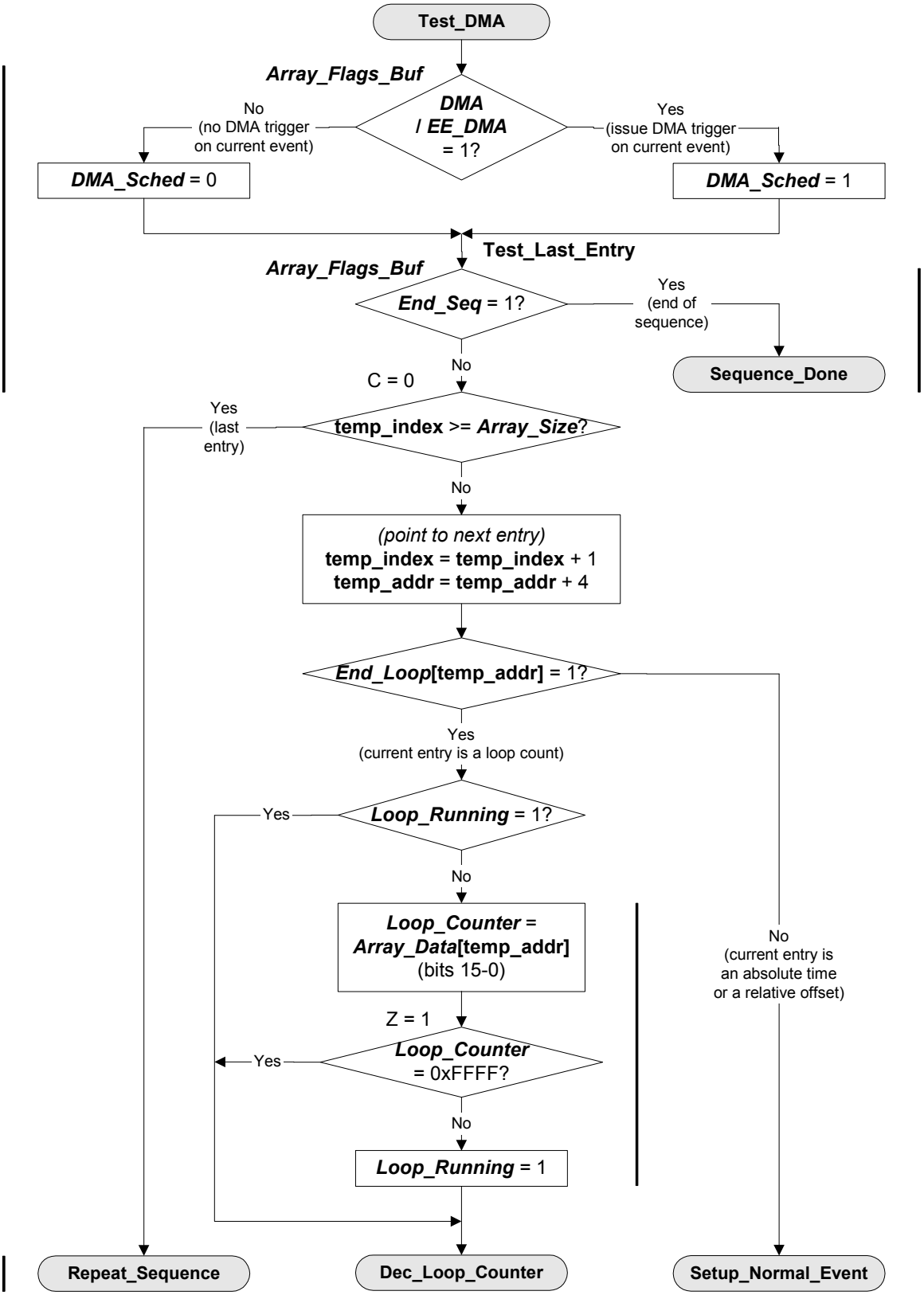

hsr = %100;
enable matches
Update
DMA_Sched = 0
Setup_End_Event

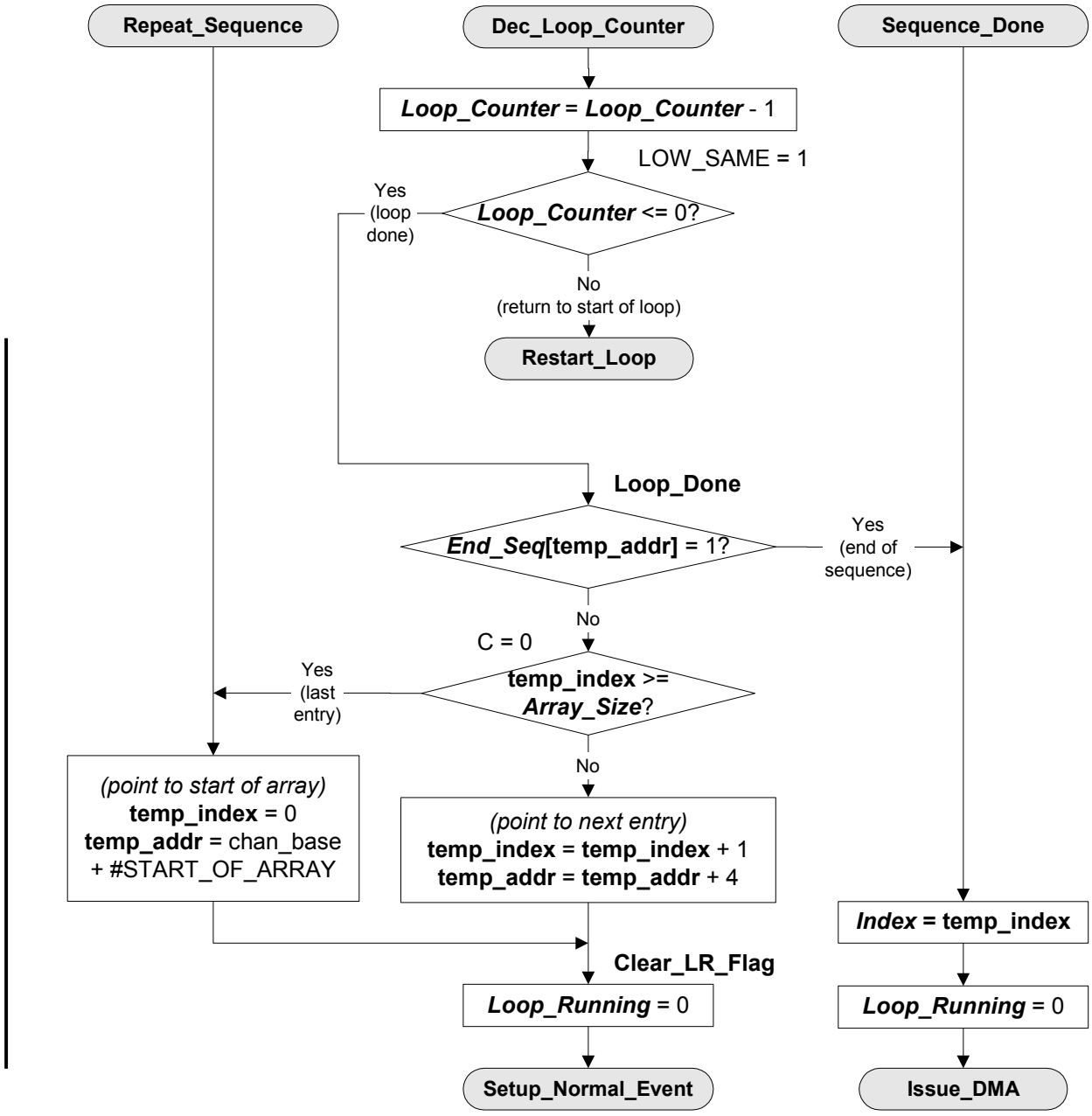

hsr = %111;
enable matches
Shutdown
Cleanup_Chan
(common.c)
EXIT

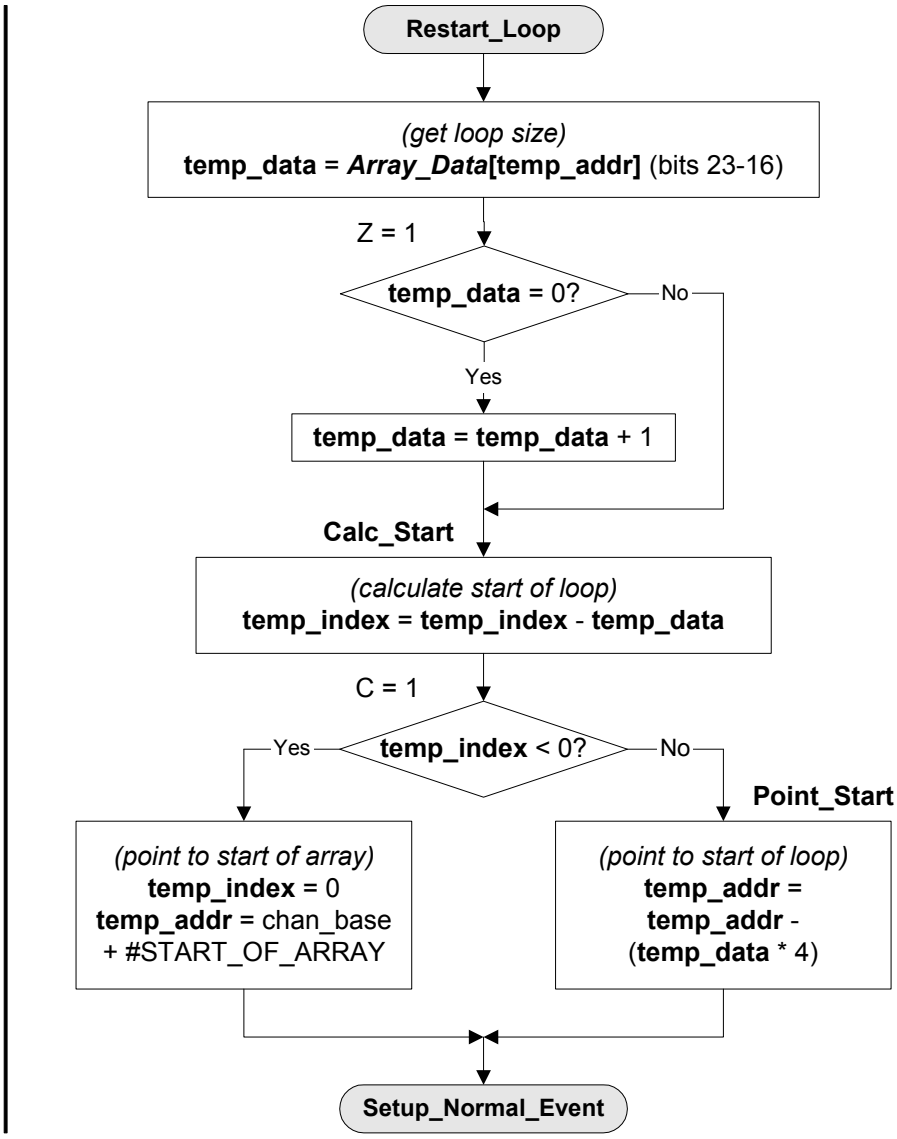

hsr = %001;
enable matches
hsr = %010;
enable matches
hsr = %011;
enable matches
Unused_HSR
Global_Exception
(common.c)

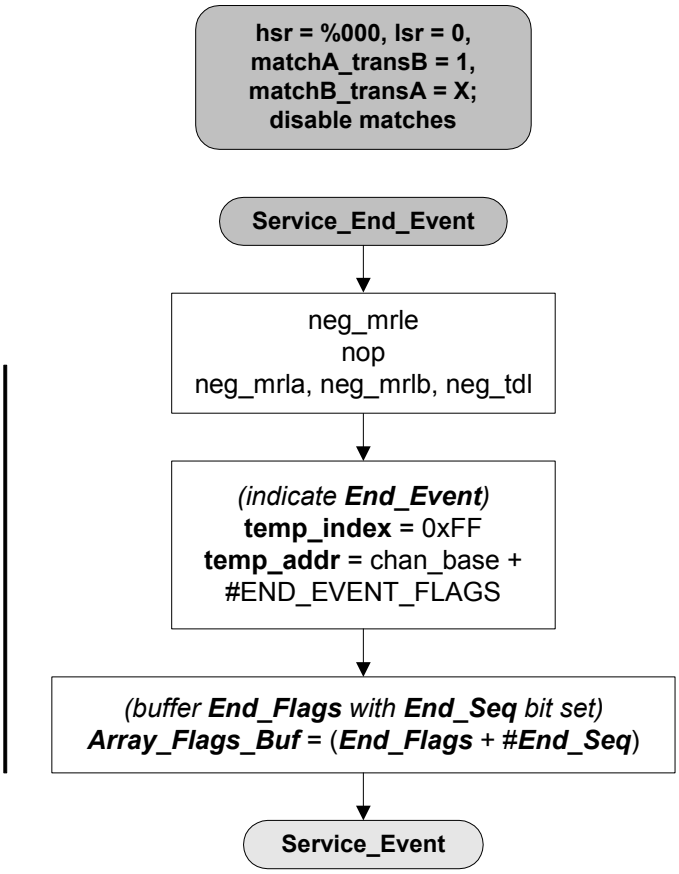

hsr = %000, lsr = 1;
enable matches
Link
neg_lsr
Request

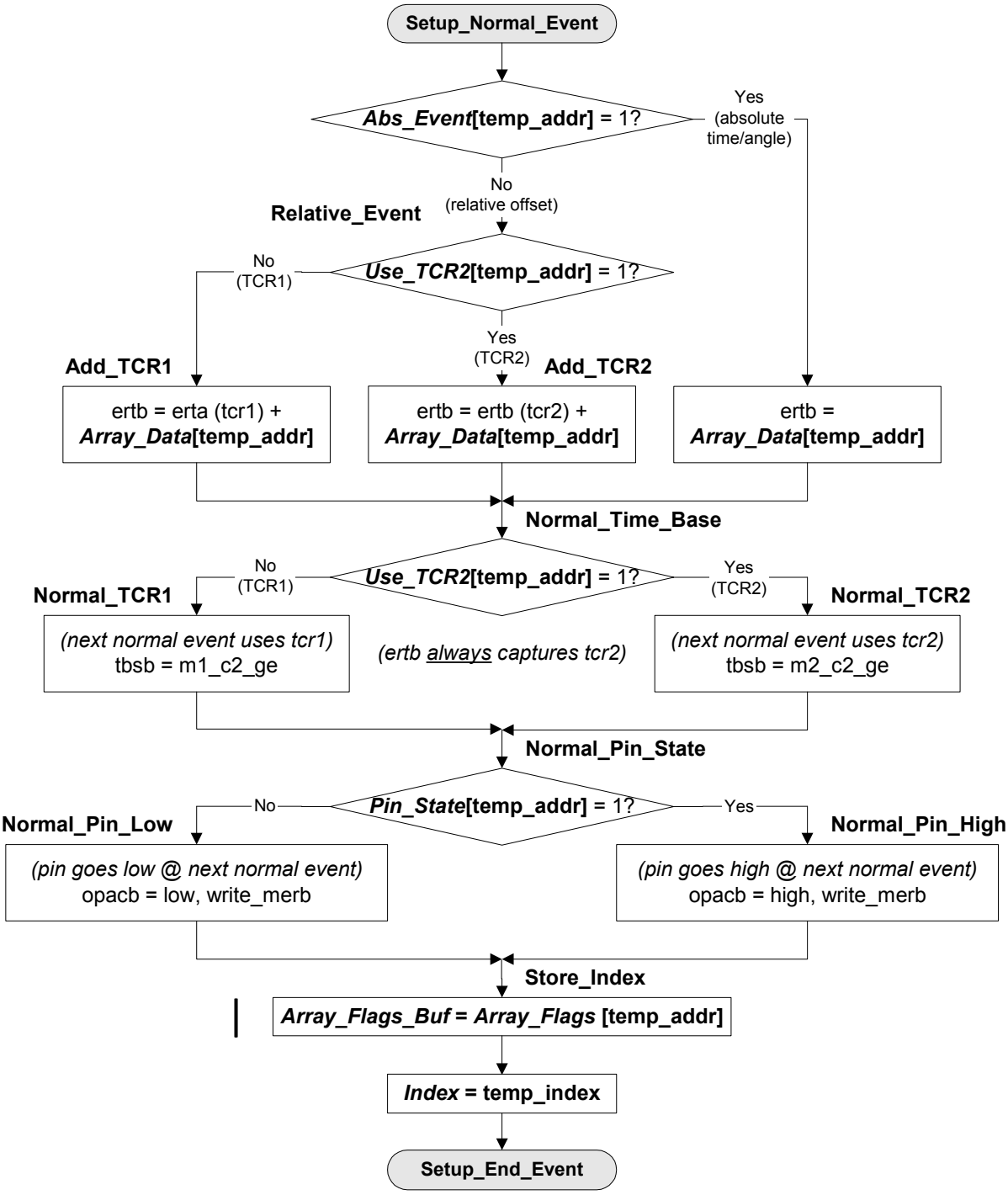

hsr = %000, lsr = 0,
matchA_transB = 0,
matchB_transA = 1;
disable matches
Service_Event
Normal_Event
neg_mrle
nop
neg_mrlb, neg_tdl
temp_index = Index
(calculate array address)
temp_addr = (temp_index * 2) +
chan_base + #START_OF_ARRAY
Array_Flags_Buf
Store_Actual
/ EE_Store_Actual
= 1?
No
(don't store
actual event
time/angle)
Yes
(store
actual event
time/angle)
Array_Flags_Buf
Use_TCR2
/ EE_Use_TCR2
= 1?
No
Event_TCR1
FM0
Event_
TCR1_Store_TCR2
= 1?
No
Yes
Event_TCR2
FM1
Event_
TCR2_Store_TCR2
= 1?
Yes
No
Yes
Store_TCR1
(store actual event in TCR1 units)
Array_Data [temp_addr] = erta
Store_TCR2
(store actual event in TCR2 units)
Array_Data [temp_addr] = ertb
Test_Intr
Array_Flags_Buf
Interrupt
/ EE_Interrupt
= 1?
Yes
(issue interrupt
on current event)
Intr_Index = temp_index
issue interrupt to CPU
No
Test_DMA

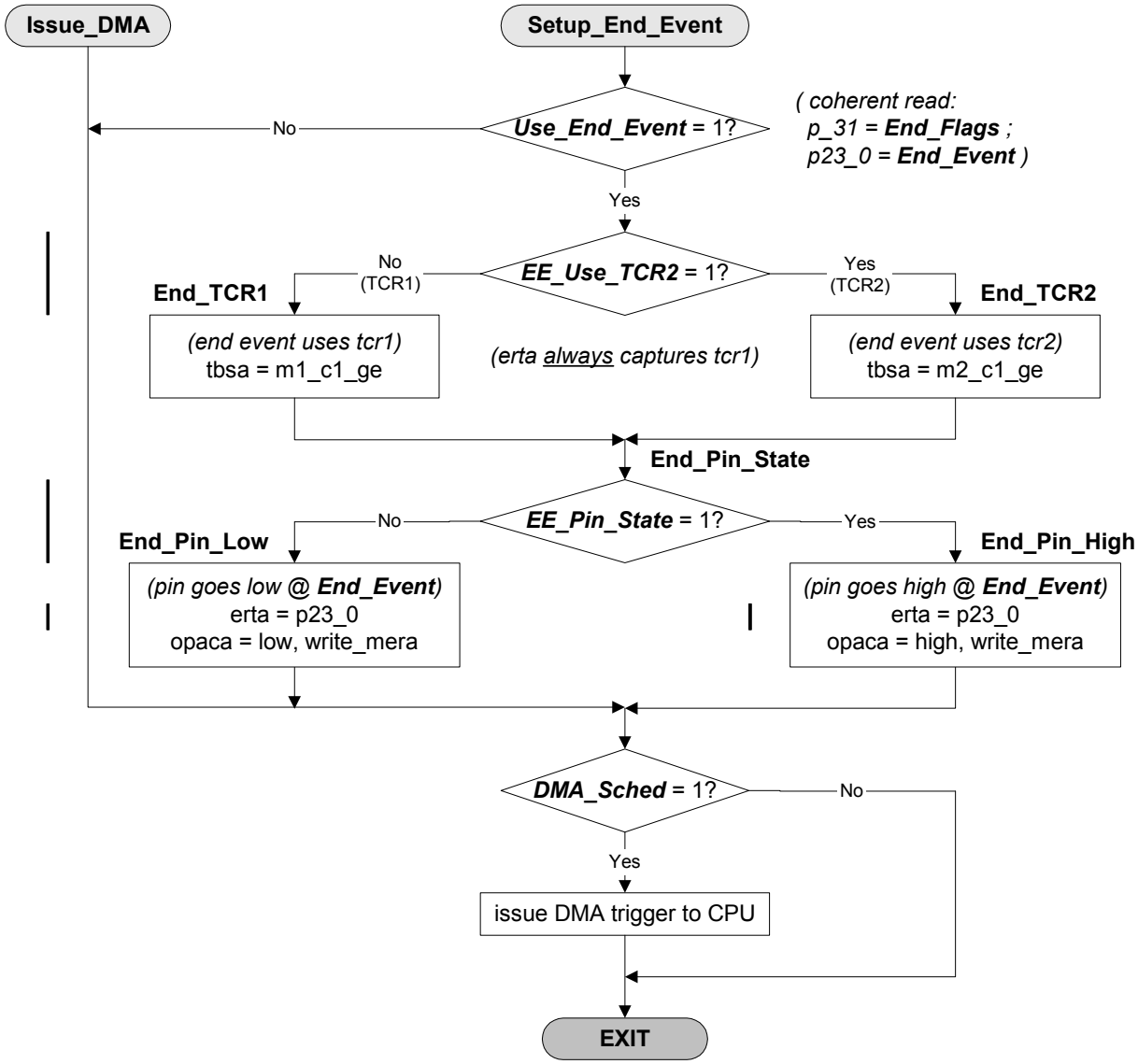

Test_DMA
Array_Flags_Buf
DMA / EE_DMA = 1?
No (no DMA trigger on current event)
Yes (issue DMA trigger on current event)
DMA_Sched = 0
DMA_Sched = 1
Test_Last_Entry
Array_Flags_Buf
End_Seq = 1?
Yes (end of sequence)
Sequence_Done
No
C = 0
Yes (last entry)
temp_index >= Array_Size?
No
(point to next entry)
temp_index = temp_index + 1
temp_addr = temp_addr + 4
End_Loop[temp_addr] = 1?
Yes (current entry is a loop count)
Yes
Loop_Running = 1?
No
Loop_Counter = Array_Data[temp_addr] (bits 15-0)
No (current entry is an absolute time or a relative offset)
Z = 1
Yes
Loop_Counter = 0xFFFF?
No
Loop_Running = 1
Repeat_Sequence
Dec_Loop_Counter
Setup_Normal_Event


Repeat_Sequence
Dec_Loop_Counter
Sequence_Done
Loop_Counter = Loop_Counter - 1
LOW_SAME = 1
Yes (loop done)
Loop_Counter <= 0?
No (return to start of loop)
Restart_Loop
Loop_Done
End_Seq[temp_addr] = 1?
Yes (end of sequence)
No
C = 0
Yes (last entry)
temp_index >= Array_Size?
No
(point to start of array)
temp_index = 0
temp_addr = chan_base + #START_OF_ARRAY
(point to next entry)
temp_index = temp_index + 1
temp_addr = temp_addr + 4
Index = temp_index
Clear_LR_Flag
Loop_Running = 0
Loop_Running = 0
Setup_Normal_Event
Issue_DMA


Restart_Loop
(get loop size)
temp_data = Array_Data[temp_addr] (bits 23-16)
Z = 1
temp_data = 0?
No
Yes
temp_data = temp_data + 1
Calc_Start
(calculate start of loop)
temp_index = temp_index - temp_data
C = 1
Yes
temp_index < 0?
No
Point_Start
(point to start of array)
temp_index = 0
temp_addr = chan_base
+ #START_OF_ARRAY
(point to start of loop)
temp_addr =
temp_addr -
(temp_data * 4)
Setup_Normal_Event


hsr = %000, lsr = 0,
matchA_transB = 1,
matchB_transA = X;
disable matches
Service_End_Event
neg_mrle
nop
neg_mrla, neg_mrlb, neg_tdl
(indicate End_Event)
temp_index = 0xFF
temp_addr = chan_base +
#END_EVENT_FLAGS
(buffer End_Flags with End_Seq bit set)
Array_Flags_Buf = (End_Flags + #End_Seq)
Service_Event


Setup_Normal_Event
Abs_Event[temp_addr] = 1?
Yes (absolute time/angle)
No (relative offset)
Relative_Event
Use_TCR2[temp_addr] = 1?
No (TCR1)
Yes (TCR2)
Add_TCR1
Add_TCR2
ertb = erta (tcr1) + Array_Data[temp_addr]
ertb = ertb (tcr2) + Array_Data[temp_addr]
ertb = Array_Data[temp_addr]
Normal_Time_Base
Use_TCR2[temp_addr] = 1?
No (TCR1)
Yes (TCR2)
Normal_TCR1
Normal_TCR2
(next normal event uses tcr1)
tbsb = m1_c2_ge
(ertb always captures tcr2)
(next normal event uses tcr2)
tbsb = m2_c2_ge
Normal_Pin_State
Pin_State[temp_addr] = 1?
No
Yes
Normal_Pin_Low
Normal_Pin_High
(pin goes low @ next normal event)
opacb = low, write_merb
(pin goes high @ next normal event)
opacb = high, write_merb
Store_Index
Array_Flags_Buf = Array_Flags [temp_addr]
Index = temp_index
Setup_End_Event


Issue_DMA
Setup_End_Event
Use_End_Event = 1?
No
Yes
(coherent read:
p_31 = End_Flags ;
p23_0 = End_Event)
EE_Use_TCR2 = 1?
No (TCR1)
Yes (TCR2)
End_TCR1
End_TCR2
(end event uses tcr1)
tbsa = m1_c1_ge
(erta always captures tcr1)
(end event uses tcr2)
tbsa = m2_c1_ge
End_Pin_State
EE_Pin_State = 1?
No
Yes
End_Pin_Low
End_Pin_High
(pin goes low @ End_Event)
erta = p23_0
opaca = low, write_mera
(pin goes high @ End_Event)
erta = p23_0
opaca = high, write_mera
DMA_Sched = 1?
No
Yes
issue DMA trigger to CPU
EXIT

6. Revision Log

6.1 Document Revision Numbering

Each microcode document is assigned a revision number. The numbers are assigned according to the following scheme (used for all documents after May, 2003):

Rev x.y

x identifies the microcode, where

1 represents the original release of microcode
2 represents the first release of changed microcode
3 represents the second change to microcode etc.

y identifies the document, where

0 represents the original release of documentation for this microcode
1 represents the first document change for the same microcode
2 represents the second change to the document for the same microcode

6.2 Revision History

Revision	Date	Record	Author
1.0 (SCR 3082)	04-16-04	Initial release of documentation.	Warren Donley
2.0 (SCR 3337)	05-15-04	- Changed ***Use_TCR2*** from a global Function Mode bit to a bit associated with each individual array entry. - Changed mode to Either Match (Blocking) so that normal events are set up with ERTB and ***End_Time*** events are set up with ERTA.	Warren Donley
3.0 (SCR 3639)	06-30-04	Fixed bug in **Request** HSR that caused ***Start_Time*** to be set up incorrectly when using TCR1.	Warren Donley
4.0 (SCR 3738)	08-20-04	Added option to issue DMA trigger to CPU.	Steven Hughes
5.0 (SCR 3936)	02-02-05	- Converted C-code to assembly language. - Combined ***Index*** and ***Array_Size*** in same parameter to save RAM space. - Renamed various RAM parameters to indicate	Warren Donley

		that they may represent either a time or an angle value (***Start_Time*** --> ***Start_Event***, etc).	
6.0 (SCR 3957)	03-18-05	- Stored actual time/angle of output events. - Buffered ***Interrupt***, ***DMA***, ***End_Seq*** and ***Store_Actual*** bits. - Changed end-of-array method. - Allowed internal loops to be infinite. - Treated links as **Request** HSRs. - Fixed coherency issues. - Replaced ***Start_Loop*** bit with **Loop Size**. - Allowed ***Start_Event*** to be a pointer.	Warren Donley
7.0 (SCR 4087)	04-20-05	Made changes to work with compiler build 147 or later.	Warren Donley
7.1	06-13-05	Added information to thread execution times.	Warren Donley