Application Report
SPNA130A–September 2011

# System Performance Improvements by Leveraging the High-End Timer Transfer Unit on Hercules™ ARM® Safety MCUs

*Brian Fortman*

**ABSTRACT**

This application report shows how the high-end timer transfer unit (HTU), a local DMA on the TMS570 and RM4x devices that is dedicated to the extremely versatile programmable timer co-processor (NHET), can be used to offload tasks from the main CPU by doing transfers between the main memory and the NHET. It shows how to set up the HTU, covers specific details that need to be taken into account when using the HTU, and lists the benefits of using HTU for data transfers instead of the CPU.

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/spna130.

**Contents**



**List of Figures**



## 1 System Performance Considerations

Many microcontroller-based applications need to generate pulse width modulated (PWM) outputs, render complicated pulse patterns, measure the period or pulse width of an incoming signal and much more. The HET (also NHET, N2HET) is a programmable timer co-processor on Hercules MCUs that helps to reduce the amount of processing the main CPU has to do for common timer/capture tasks and also performs some tasks that cannot be done by a traditional hardware timer. However, in most applications there is still a lot of data that needs to be transferred between the timer and the main memory by the CPU, both for input and output signals, thus reducing the amount of time the CPU has for other tasks. The HTU is very handy for handling these transfers. In order to help the following discussion, Figure 1 shows a simplified block diagram of a TMS570 or RM4x safety microcontroller architecture. For the block diagram that applies to your device, see the device-specific data sheet.

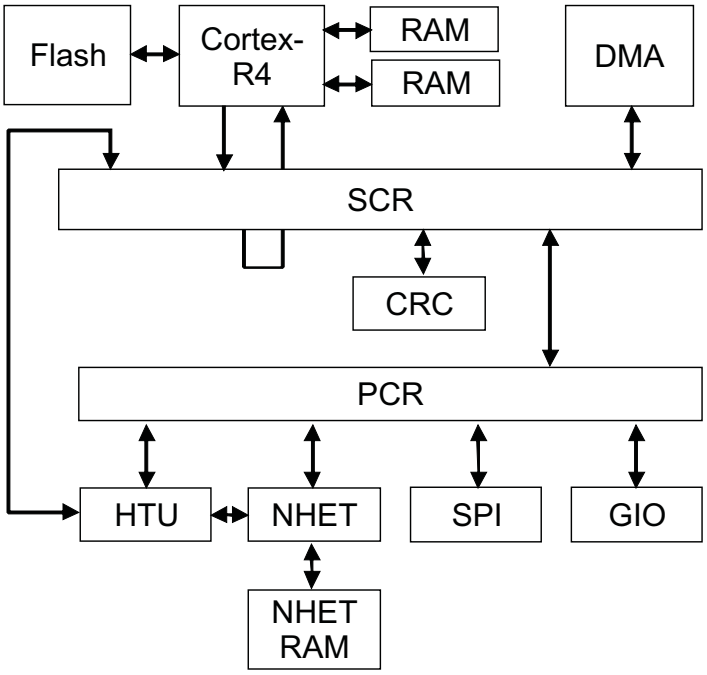


**Figure 1. Typical TMS570/RM4x Device Block diagram**

If the CPU has to access the NHET RAM to update values or retrieve data, the path is via the switched central resource (SCR) to the peripheral central resource (PCR). In many cases these busses are running at different frequencies, which means that a single access can take multiple cycles. The greater the number of accesses the CPU has to make, the more impact they will have on the overall system performance.

So, why not use the system DMA for doing all those data transfers to the timer? It can be done and it would offload some of the dead time of the CPU for other things, but both the CPU and the DMA share the same *single bus interface to the PCR*. So if both do peripheral accesses there will be contention and arbitration between the two that will slow down one or the other.

The HTU works around this bottleneck by providing direct access to the NHET RAM and a master port into the SCR for direct access to the R-4 TCM RAM. The SCR allows concurrent transactions by different masters to the different slaves of the system. In the example in Figure 1, there could be a CPU access to the CRC module, a DMA access to the peripherals, and a HTU transfer to the main RAM of the CPU, all at the same time. The other benefit is that the data needed or updated by the CPU resides locally in the CPU RA, which often times is a single cycle access. So the CPU can get to the relevant data much faster and can free accesses to the PCR for the DMA, thus increasing the overall system performance.

On a side note, one might wonder by looking at Figure 1, why the HTU has no direct access to the NHET RAM. Although the NHET architecture allows multiple accesses to different RAM locations at the same time, there is still the chance that both the NHET and the HTU may contend for the same memory location in the same clock cycle. This scenario has been taken care of by adding an arbiter between the different masters who are able to access the NHET RAM. The NHET will always get priority so that the NHET program execution is not stalled. The Hercules architecture has been designed to provide very high performance for most access scenarios.

## 2 HTU Features

Now that some basic understanding of how the HTU could benefit the overall system performance has been established, it is time to look a bit closer at the features of the HTU and how it works. This application report will not address every single HTU feature, but will provide a good foundation for solving more complicated application scenarios.

As mentioned before, the HTU is a local DMA to the NHET module and allows for the transfer of data to and from the NHET RAM. It works similar to the system DMA as there are up to eight request lines (HTUREQ[x]) connected between the NHET and the HTU. Several of the NHET instructions can generate trigger events on these request lines at the occurrence of special events. The requests are tied to control packets that allow programming the transfer direction, the source and destination address and how much data should be transferred. A triggered request then kicks off a single or multiple data transfer to or from the NHET memory based on the information stored in the control packet. The control packets are called Double Control Packet (DCP) because they provide the capability to set up two buffers for the bi-directional transfer of data. For example, the HTU could transfer data to one buffer while the CPU works on the other one. Once the HTU has filled its buffer it can automatically switch to the other one and start filling it with new data.

## 3 Setup of NHET and HTU

You can use the HTU to transfer measurement data from the NHET to the main memory into a single buffer. For this, you create a simple NHET program that measures the period of an input signal with the help of the PCNT instruction.

### 3.1 NHET Program

```
L01   PCNT     {next=L02, reqnum=0, request=GENREQ, irq=OFF, type=FALL2FALL, pin=10, period=0,
               data=0, hr_data=0}
L02   BR       {next=L01, cond_addr=L01, event=NOCOND}
```

Two parameters in the PCNT instruction are of interest. The first is the *reqnum* parameter, which tells the NHET the HTU request signal to trigger when new measurement data is available in the data field of the instruction. The other parameter is *request*, which specifies the type of request should be generated. There are different types, but for this example, stick with the standard request that will be generated.

### 3.2 NHET Setup

Once the program has been loaded into the NHET RAM, the NHET needs to be configured. One of the registers that need to be programmed is the *Request Destination Select Register* (HETREQDS). This one lets you choose to generate a HTU or DMA request or both. For this example, you just want to generate a HTU request and the request line selected in the PCNT instruction is HTUREQ[0], so you need to program bits *TDBS0* and *TDS0* to '0'. The other register is the *Request Enable Set Register* (HETREQENS). This register enables the request to generate the trigger to the HTU. Bit *REQENA0* needs to be set to '1' to enable request HTUREQ[0].

```
void hetInit(void){
…
HETREQDS      &= 0xFFFEFFFE;    /* Set bits TDBS0 and TDS0 to 0     */
HETREQENS     |= 0x00000001;    /* Enable request HTUREQ[0]         */
…
}
```

### 3.3 HTU Setup

Next, you have to configure the HTU. Before you do that, you should define how big the buffer is that will store the measured data. For this example, you want to capture 10 entries.

In the HTU, you need to set up a few registers and the control packet corresponding to the selected HTUREQ signal.

For the control packet, you have to define the start address of the buffer where the measured data will be stored, the address of the NHET data field of the PCNT instruction where the measured data will be read from, and how many elements and frames should be transferred. Each new HTU request will trigger a frame transmission. Each frame can have one or multiple elements. So, for example, if only one field from a single instruction should be transmitted, the element count in a frame would be one. If multiple fields from multiple instructions should be transferred with a single HTU request, the element count should correspond to the number of fields that need to be transferred. Figure 2 shows an example of elements and frames.

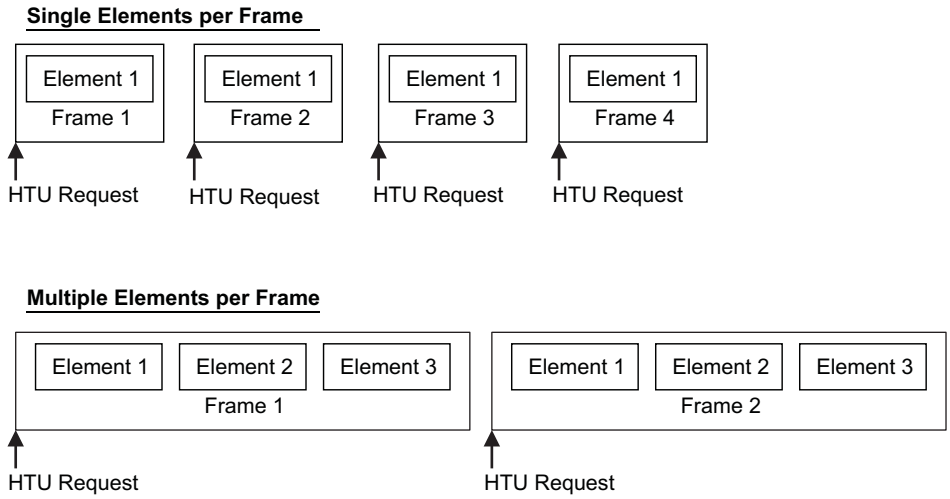


**Figure 2. Element and Frame Count example**

An element can be either 32-bit or 64-bit wide.

In this example, you only have the PCNT data field to move into the buffer, so you have to set up the element count per frame as ‘1’ and since the buffer is 10 entries deep, you need to set up the frame count as ‘10’. The data field of the PCNT instruction is at address 0x8 from the NHET RAM. Note that since the HTU has direct access to the NHET memory, you do not have to specify the 32-bit address from an ARM CPU point of view. To finish the control packet setup, you need to specify the destination address of where the data should be transferred, which corresponds to the start address of the buffer.

```
…
unsigned int bufferA[10];
…
void HTU_init(void){
…
htuCPRAM ->DCP[0].ITCOUNT = 0x0001000A;    /* DCP0 CPA element count = 1, frame count = 10 */
htuCPRAM ->DCP[0].IHADDRCT = 0x00000008; /* DCP0 CPA DIR = NHET to main memory          */
            /*          SIZE = 32-bit                        */
            /*          ADDMH = 16 bytes (not relevant for  */
            /*          this example)                        */
            /*          ADDFM = post-increment main memory  */
            /*          TMBA = one shot buffer A             */
            /*          TMBB = one shot buffer B (not rel.) */
            /*          IHADDR = 0x2 => 0x8 PCNT data field */
htuCPRAM ->DCP[0].IFADDRA = (unsigned int)bufferA;
            /* DCP0 CPA start address of destination buffer */
…
}
```

The buffer is set up for *one shot* mode. This means it will be filled once and when full the HTU will not transfer any additional data although the NHET will still generate requests to the HTU.

## 4 Example Project

In order to try out the above, an example project can be downloaded at http://www.ti.com/lit/zip/spna130.

The project can be configured to a single buffer in one shot mode, just like the above description, or in double buffer auto switch mode. Further, it can be configured to generate an interrupt when the buffer is full.

The configuration can be made in the *main.h* file.

```
#define BUFFERSIZE 20         /* size of the single or double buffer           */
#define INT 0                    /* 0 = interrupt disabled; 1 = interrupt enabled */
#define DOUBLEBUFFER 0        /* 0 = single buffer; 1 = double buffer          */
```

The project needs to be built every time one of the configurations is changed.

There is no need to apply an external PWM input signal to pin NHET[10], since the project takes advantage of the fact that software can set up the pin as output and toggle the pin by writing to the HETDOUT or HETDSET or HETDCLR registers and the NHET program measures back the period of the signal via the internal loopback capability of the pin structure.

The captured periods are stored in the arrays *bufferA* and *bufferB*. If interrupts are enabled, then numbers of interrupts executed can be viewed in variables *htu_highintcp0a_count* and *htu_highintcp0b_count*.

## 5 Conclusion

This small application report clarifies the benefits of using the high-end timer transfer unit. Although the example used in this document is very simple, it should convey the basic functionality of the module. Together with the flexibility of the NHET, there are many applications that can benefit by using the HTU to offload common tasks from the main CPU; thus freeing up the CPU to work on other tasks.

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265