

## **Hardware techniques for removing CPU overhead in Microcontroller applications.**

*By Graeme Clark, product marketing manager, Renesas Technology Europe*

One of the biggest challenges facing developers today is to find ways to increase real time system performance without impacting clock speed (due to EMC constraints), software complexity and, of course, cost. For developers of microcontroller applications this has not been such a big issue to date. However, it's starting to become more and more of an issue even for the developers of traditional 8 & 16 bit microcontroller applications.

This requirement is being driven by a number of conflicting requirements:

- The increasing trend towards using operating systems or schedulers in applications using 8-bit microcontrollers. This adds greatly to the system overhead in handling real time events.
- The addition of more and more functions onto what used to be simple applications, especially in the areas of user interface and communications, resulted in less processor time being available for the real time tasks.
- The requirement to simplify testing and hence increase the reliability of a system.

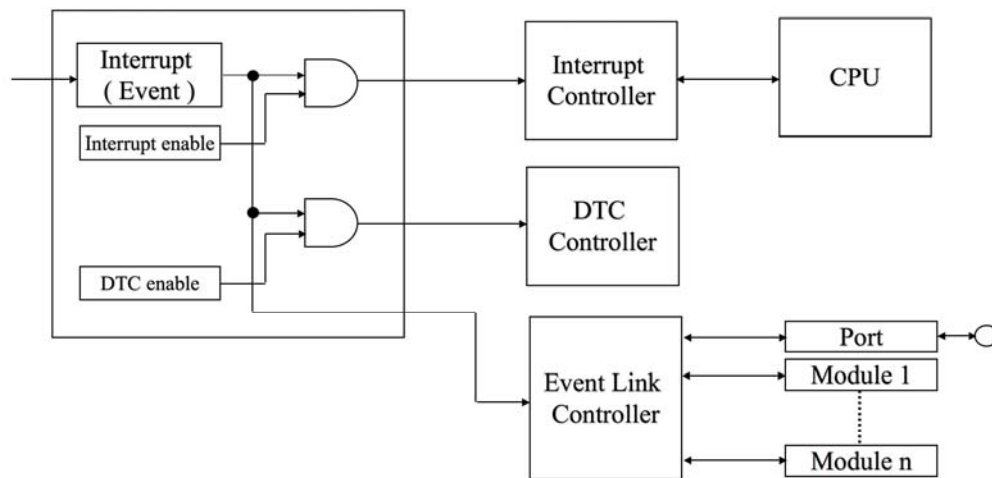
With this in mind, a lot of research is being conducted to find solutions to these problems, even looking at the fundamental ways that microcontrollers handle real time events, and whether these can be improved.

One possible solution is to reinvent the basic real time structure of a microcontroller, and try and move more of the fundamental real time response issues, from software where they currently reside, to a more hardware based

solution. This can result in a much faster response time to real time events, as well as a reduction in both software size and complexity. This allows the hardware of the chip itself to handle many of the real time events that in the past would have required CPU intervention.

We will present here how a typical implementation of this new structure can work.

The structure creates three types of “event” inside the microcontroller, generated from any external or internal interrupt source. These can include a wide variety of real-time stimuli, such as an input pin state changing, a timer timing out or a byte arriving in a serial interface.



**Figure 1 – Interrupt controller block diagram**

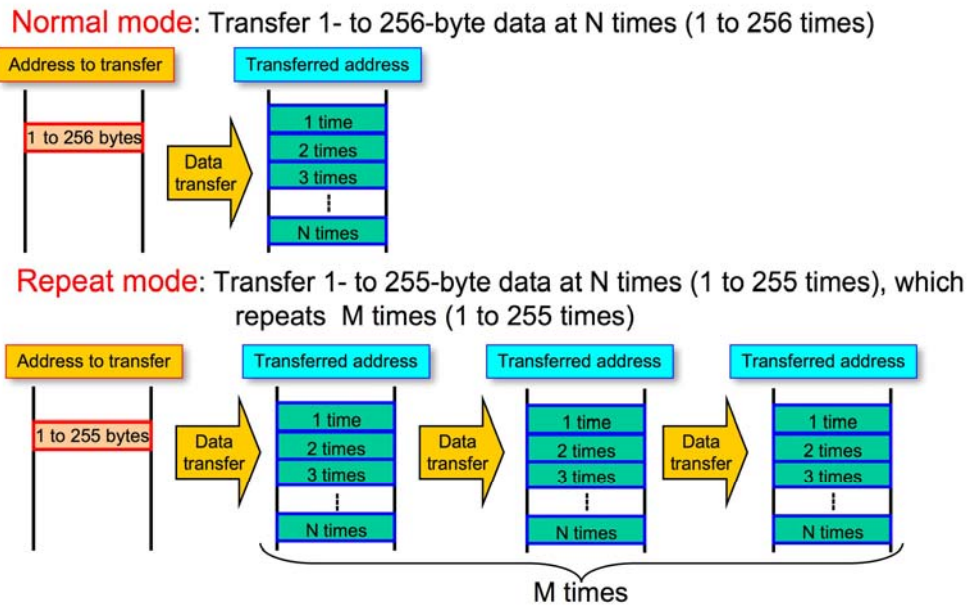
The diagram above shows the block diagram of an interrupt controller and associated logic, which can be used to handle real time events. The device implements standard interrupts, using a 4 priority level interrupt controller, however as can be seen, you also have a choice of generating an automatic data transfer using the Data Transfer Controller (DTC) instead of an interrupt, or an “Event” using the Event Link Controller (ELC). You can choose to generate any one of these or any combination of these in response to any real

time event in the micro-controller. However, the real benefit of this device is in handling such events using either the DTC or the ELC, as these are then handled without any software or CPU time, and so being executed both faster, and without any software overhead.

The DTC provides much the same functions as a DMA controller, as it is designed to allow the transfer of one or more bytes between memory and a peripheral or peripheral and memory, the transfer being triggered by an event on the chip. However it's designed to be both much cheaper to implement and to be much more flexible than a DMA controller.

It achieves this by using the CPU's logic to make the transfer, rather than a large block of dedicated hardware. The advantage of this is that it's programmable, it holds its set up information in a small block of on chip SRAM and so the DTC controller can be used not just to create one or two channels of data transfer, but 10 or 20 if required. The disadvantage is that the CPU is stopped for a few cycles while this transfer takes place.

The DTC can also operate in a number of modes. These are shown below.

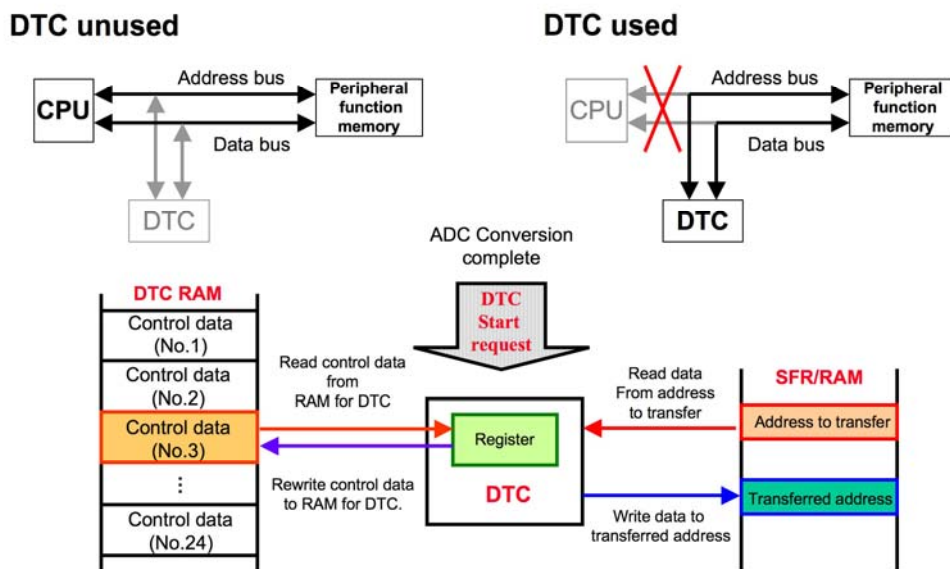


**Figure 2 – DTC can operate in a number of modes.**

The DTC can transfer 1 byte or more than one byte, between a peripheral and memory or memory and peripheral up to 256 times. The address in memory can be the same address or it can be incremented or decremented, so creating a buffer structure. At the end of the transfer, the DTC can generate an interrupt, to tell the CPU the data is ready, or in fact, it can trigger a second DTC transfer. In fact this can be used to chain a number of transfers together if required.

The DTC can also be placed into repeat mode, where it will repeat the transfer an additional number of times.

In this case, we are using the ADC in scan mode, reading 4 ADC channels and storing the result in separate result registers. After the 4<sup>th</sup> reading is complete, the ADC module generates an interrupt request, which is used to initiate a DTC transfer. Below you can see the sequence of events that occurs.



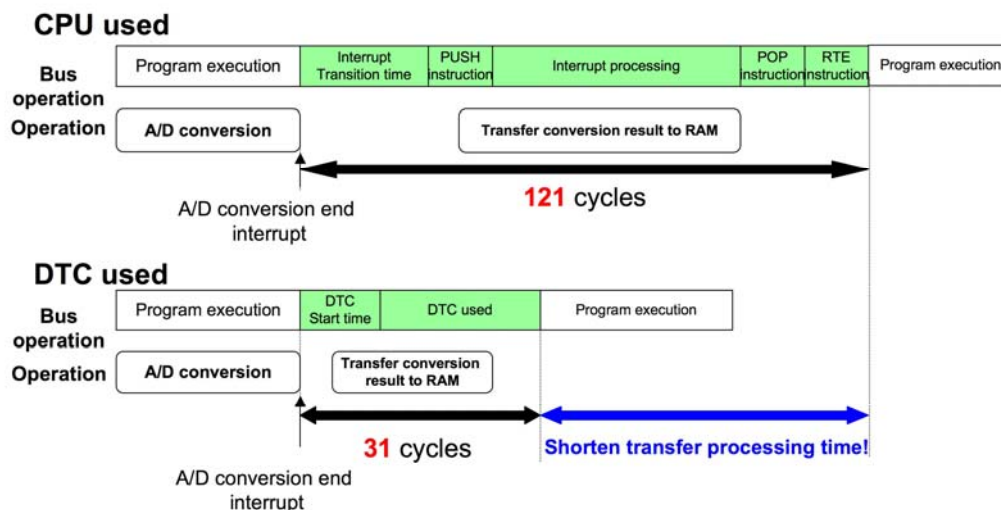
**Figure 3 – DTC transfer example**

After the CPU finishes executing the current instruction, the DTC controller will take over control of the bus. The DTC controller reads the control data held in SRAM corresponding to the ADC (12 bytes per DTC channel) which

contains the start and destination address of the data as well as how many bytes and the type of transfer.

Then it initiates the transfer of 8 bytes into the SRAM, incrementing the address for each transfer.

The DTC controller then writes back the status into the SRAM and releases the CPU to execute the next instruction. The whole transfer process in this case takes 31 clock states, from start to finish. If we made this transfer using an interrupt, when you take into account for the interrupt response time, the time required to transfer the data, and the return from interrupt, the CPU would take a minimum of 131 cycles to do the transfer. So, the Data Transfer Controllers is almost 4 times faster. We also have the added bonus of requiring no software to make the transfer, so saving a little code space. You can see the detail of this transfer in the next diagram.



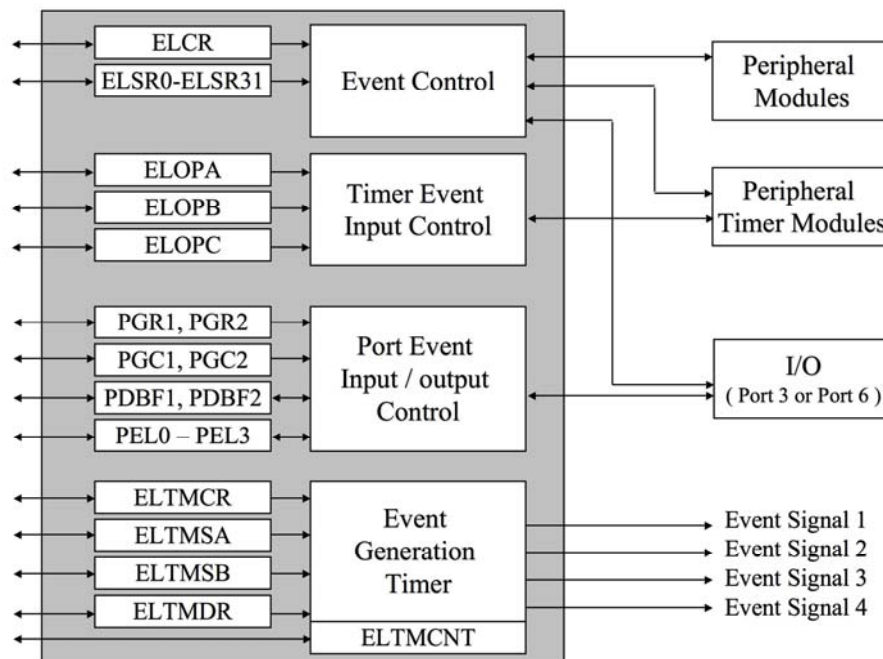
**Figure 4 – CPU v DTC timing example**

For most applications the flexibility of the DTC provides a perfect compromise, between speed, flexibility and, of course, device cost. You can create automated transfers between any peripheral and memory almost without limit.

The Data Transfer Controller provides a method for automating the transfer of data between peripherals and memory. The second new method of handling device events, uses the Event Link Controller (ELC).

The ELC is designed to allow device events to automatically cause actions on the chip. For instance, the ELC can start a timer or an ADC conversion to start, or even an I/O pin to toggle, completely automatically.

The main functional blocks of an example ELC are shown in the next diagram.



**Figure 5 - Event Link Controller**

The ELC comprises 4 main modules, the Peripheral Event Controller for the peripherals, the Timer Event Input controller, the Port Event Controller and the Event Generation Timer.

The Peripheral Event Controller allows the user to choose which peripheral is controlled by which event. This allows, for instance, the overflow from a 16 bit timer to start the ADC. The Timer Event Input controller selects what each

timer will do when it is triggered by an event. A list of all the possible actions that can be generated in response to an event is shown below.

<b>Module</b>	<b>Operation when an Event is input</b>		
<b>Timer RA</b>	Each timer operates differently depending on the setting of the relevant event		
<b>Timer RB</b>	link option setting register as below.		
<b>Timer RC</b>	* Start Counting when an event is input		
<b>Timer RD</b>	* Counts the input events		
<b>Timer RG</b>	* Performs input capture when an event is input		
<b>A/D Converter</b>	Starts A/D conversion when an event is input.		
<b>D/A Converter</b>	Starts D/A conversion when an event is input.		
<b>Output Ports</b>	The value of PDR ( port data register) changes when an event is input. (The value of the signal to be output from the relevant external pin changes.)	<b>Port Group</b>	The port-group operates differently depending on the settings below. * Changes the PDR value to the specified value. * Transfers the PDBF values to the PDR. * Shifts out the bit value.
		<b>Single Pin</b>	Changes the PDR value to the specified value.
<b>Input Ports</b>	When the value of the input pin changes. When an event is input	<b>Port Group</b>	Generates an Event
		<b>Single Pin</b>	Generates an Event
		<b>Port Group</b>	Transfers the signal value of the external pin to PDBF.
		<b>Single Pin</b>	Event connection impossible.
<b>CPG</b>	Switches the current operation to the low-speed on-chip oscillator operation.		
<b>DTC</b>	Starts data transfer.		
<b>Interrupt Controller</b>	Issues an interrupt request to the CPU.		

**Figure 6 – ELC actions**

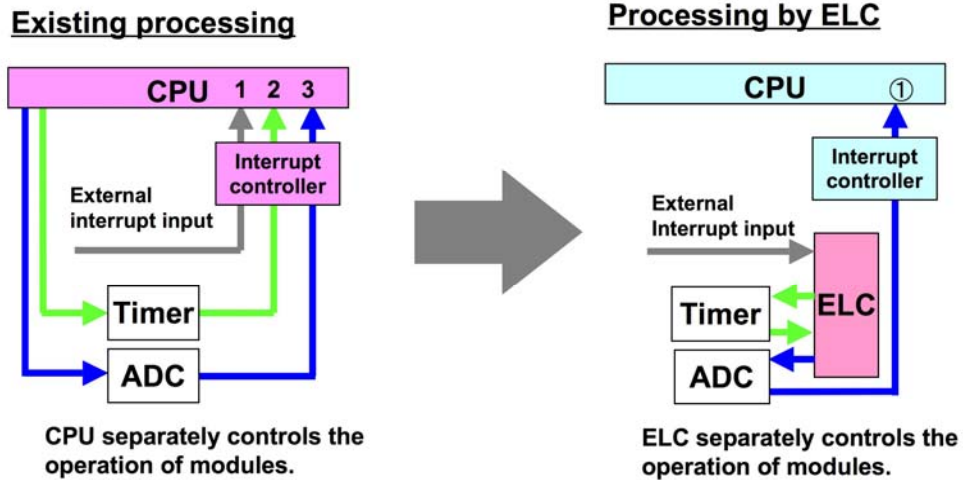
One of the most powerful features of the Event Link Controller is the ability to generate events from the I/O Ports. The port pins can be used individually or in groups of 4 pins. Each pin or group of pins can be used to generate an event, which then can control other functions on the chip. However they can also be themselves controlled by the Event link Controller. When an event is generated you can, for instance, choose to toggle a pin or group of pins, or drive them to a specific logic level. Behind each group of pins, there is also a port buffer, you can use an event to drive the contents of that buffer onto the pins, and if required, rotate the buffer ready for the next event. This type of feature is idea for generating output waveforms of various types, such as for driving stepper motors and LCD displays. You can also use this buffer to latch the input values on the pin when an event occurs.

The final functional block is the Event Generation Timer, this is a dedicated 4-channel timer that allows you to generate up to 4 regular events, each of which can be used to trigger an action on the chip. So, if you have any function that requires a regular operation, for instance if you want the ADC to sample every 50 mS, or an I/O pin to toggle every 800 nS, then you can set up the Event Timer to generate such an event.

The combination of these features allows the developer to automate a number of the real time elements in their system. An event will take 4 clock cycles from the time the event is recognised before the resulting action starts, for instance an I/O pin toggling or a timer making a capture. This feature alone enables the users to handle many real time events in hardware on the chip automatically, without the overhead of CPU intervention.

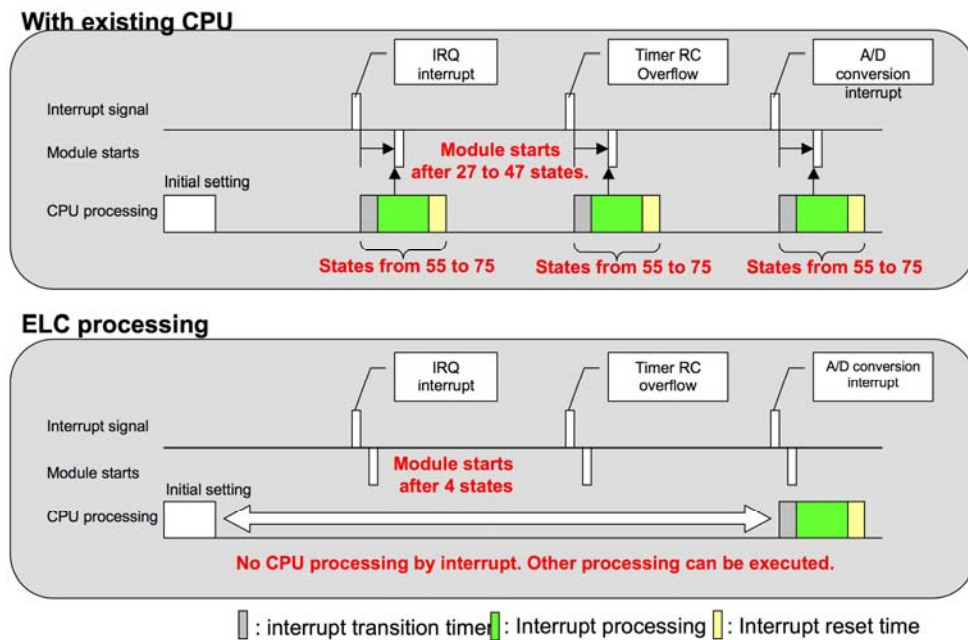
A simple example of the use of the Event Link Controller is shown below. An external interrupt, causes a timer to start, after the timer overflows, this then causes the ADC to start a conversion. When this system is implemented on a typical microcontroller, it requires 3 interrupts, one for the external interrupt, one for the timer overflow interrupt and one for the ADC end of conversion, with the resulting CPU overhead, and of course the software required for each Interrupt Service Routine.

The system will also experience some jitter, especially if the system is using an operating system, which may be handling a higher priority task when one or more of these interrupts occur.



**Figure 7 – Interrupt v ELC processing**

The whole function can be handled by the Event Link Controller. After the Event Link Controller is initialised, the whole process can be handled automatically without any CPU involvement until the complete process is finished. We can compare the process flow of using the event Link Controller against the use of interrupts in the next diagram.



**Figure 8 – Processing time comparisons**

In a typical microcontroller, after each interrupt is generated, we have to service each interrupt. This results in a delay, and depending on the software, some jitter. In a microcontroller with the Event Link Controller, each event triggers the next peripheral automatically. The CPU is only interrupted after the ADC conversion has finished.

So in this example we have no jitter, as each peripheral starts 4 clock cycles after the generating event, and is not effected at all by the condition of the CPU, whether it's handling a priority task or not. We also have the additional saving, of apart from the initialisation, there is no software required for this process so code space required for the application is reduced.

We could go even further to automate this example and reduce CPU load, by using the DTC. We could automatically transfer the data from the ADC into a buffer in SRAM. Therefore, we could choose to interrupt the CPU perhaps only after every 100 or 200 readings.

The combination of the Data Transfer Controller and the Event Link Controller means that many low level tasks, especially those based around the I/O and timers can be handled automatically without CPU intervention. This both reduces the load on the CPU and greatly reduces the system reaction time to external events.

The H8STiny family from Renesas is the first device to combine the Event Link Controller and the Data Transfer Controller. Together they can be used to remove much of the real time response requirements of the system and to automate many of the low level and repetitive tasks. The microcontroller family have been aimed at a wide range of communications and control applications.