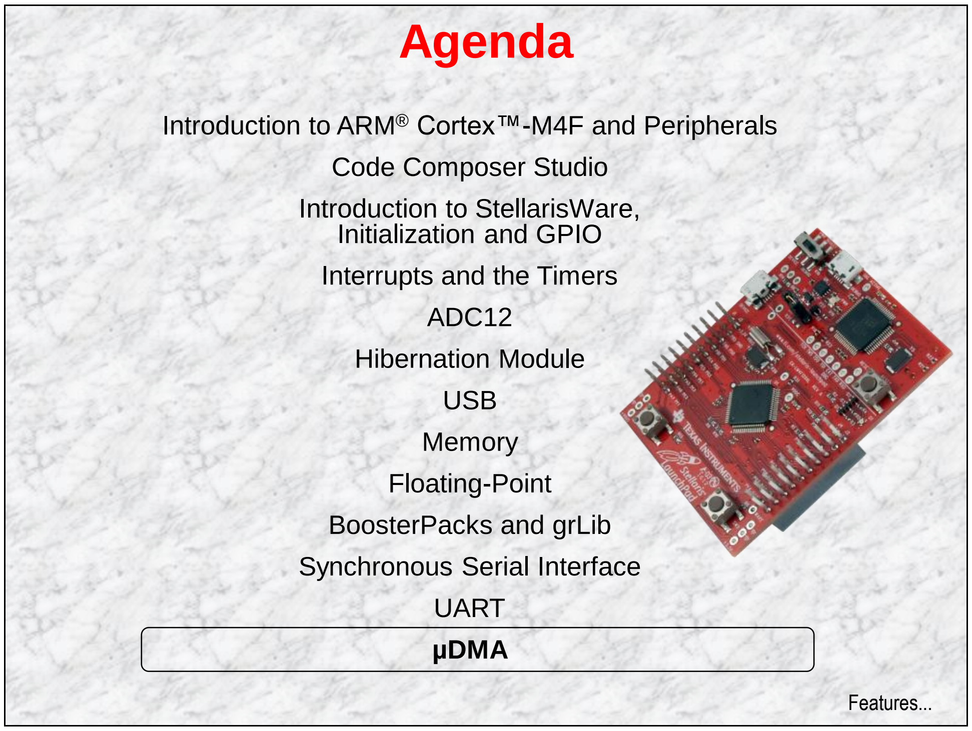
Introduction

This chapter will introduce you to the micro DMA (µDMA) peripheral on Stellaris devices. In the lab we'll experiment with the µDMA transfers in memory and to/from the UART.



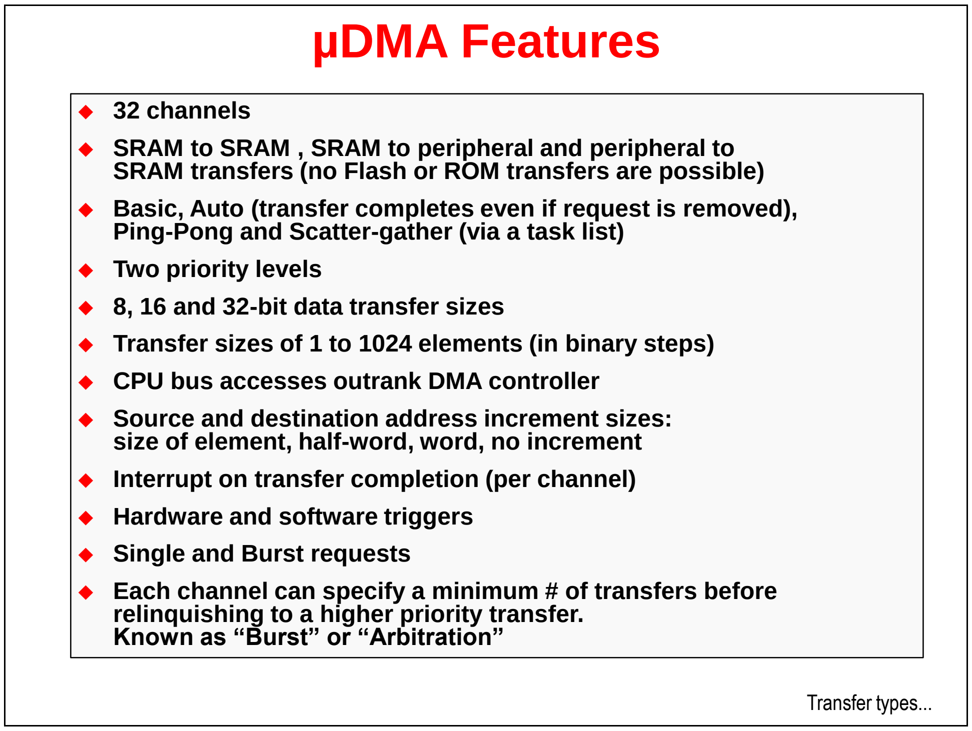
Agenda

Introduction to ARM® Cortex™-M4F and Peripherals
Code Composer Studio
Introduction to StellarisWare, Initialization and GPIO
Interrupts and the Timers
ADC12
Hibernation Module
USB
Memory
Floating-Point
BoosterPacks and grLib
Synchronous Serial Interface
UART
µDMA

Features...

Chapter Topics

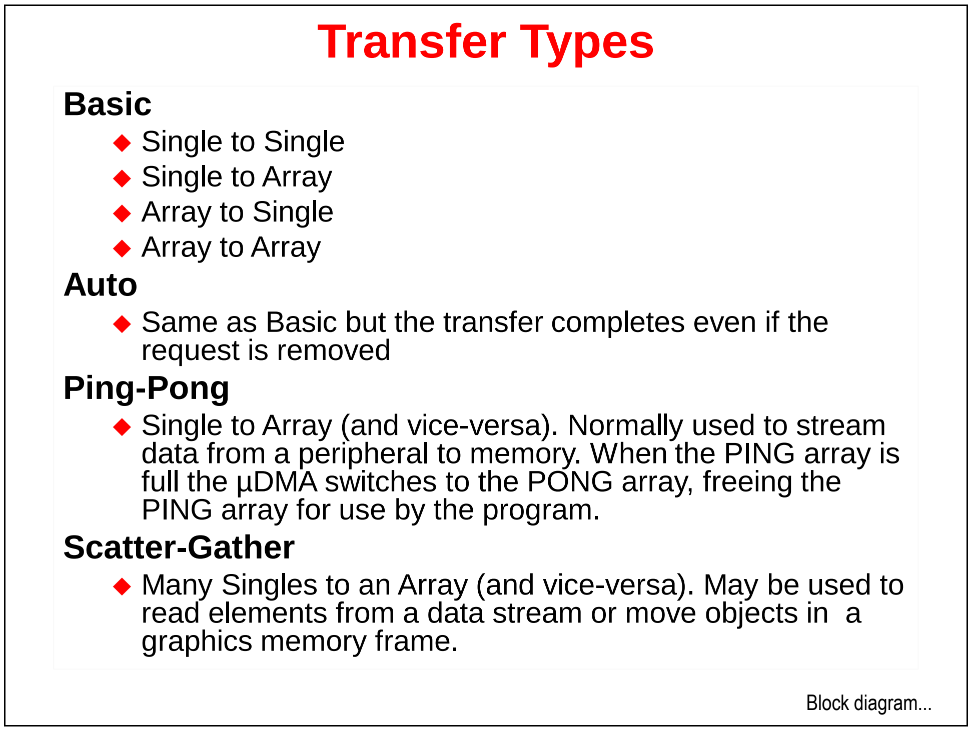
Features and Transfer Types



µDMA Features

- 32 channels
- SRAM to SRAM , SRAM to peripheral and peripheral to SRAM transfers (no Flash or ROM transfers are possible)
- Basic, Auto (transfer completes even if request is removed), Ping-Pong and Scatter-gather (via a task list)
- Two priority levels
- 8, 16 and 32-bit data transfer sizes
- Transfer sizes of 1 to 1024 elements (in binary steps)
- CPU bus accesses outrank DMA controller
- Source and destination address increment sizes: size of element, half-word, word, no increment
- Interrupt on transfer completion (per channel)
- Hardware and software triggers
- Single and Burst requests
- Each channel can specify a minimum # of transfers before relinquishing to a higher priority transfer. Known as “Burst” or “Arbitration”

Transfer types...



Transfer Types

Basic

- Single to Single
- Single to Array
- Array to Single
- Array to Array

Auto

- Same as Basic but the transfer completes even if the request is removed

Ping-Pong

- Single to Array (and vice-versa). Normally used to stream data from a peripheral to memory. When the PING array is full the µDMA switches to the PONG array, freeing the PING array for use by the program.

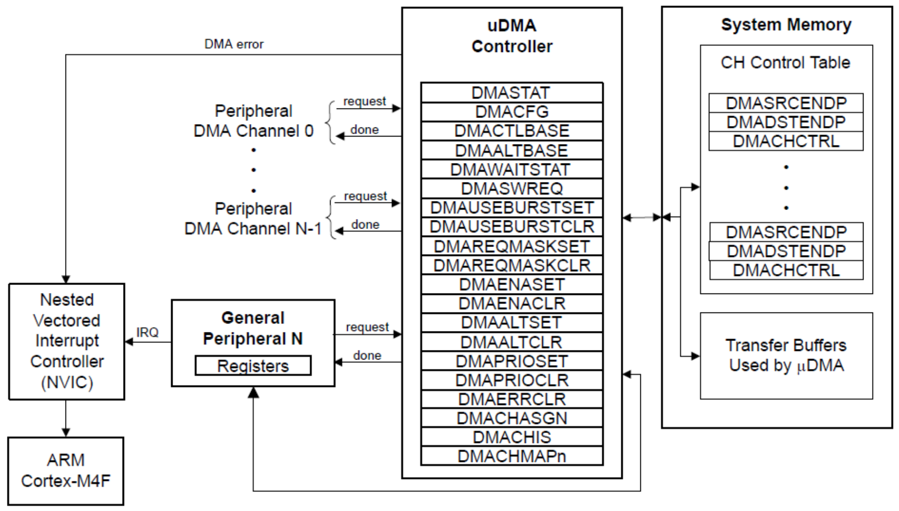
Scatter-Gather

- Many Singles to an Array (and vice-versa). May be used to read elements from a data stream or move objects in a graphics memory frame.

Block diagram...

Block Diagram and Channel Assignment

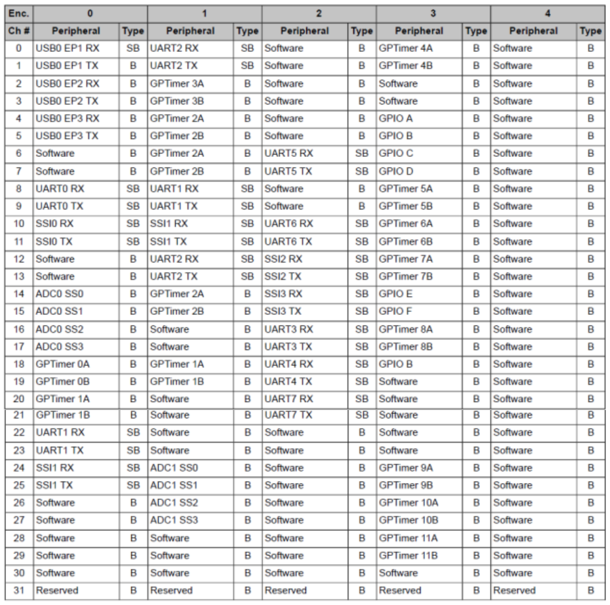
µDMA Block Diagram



Channels...

µDMA Channels

- Each channel has 5 possible assignments made in the DMACHMAPn register

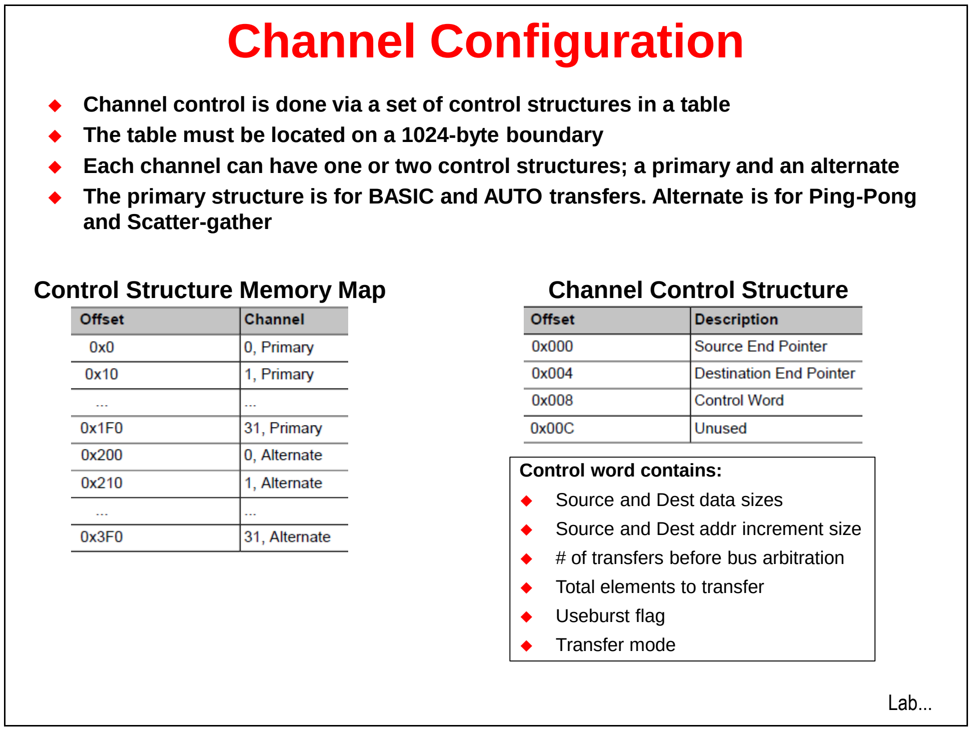


Enc.	0		1		2		3		4	
Ch #	Peripheral	Type	Peripheral	Type	Peripheral	Type	Peripheral	Type	Peripheral	Type
0	USB0 EP1 RX	SB	UART2 RX	SB	Software	B	GPTimer 4A	B	Software	B
1	USB0 EP1 TX	B	UART2 TX	SB	Software	B	GPTimer 4B	B	Software	B
2	USB0 EP2 RX	B	GPTimer 3A	B	Software	B	Software	B	Software	B
3	USB0 EP2 TX	B	GPTimer 3B	B	Software	B	Software	B	Software	B
4	USB0 EP3 RX	B	GPTimer 2A	B	Software	B	GPIO A	B	Software	B
5	USB0 EP3 TX	B	GPTimer 2B	B	Software	B	GPIO B	B	Software	B
6	Software	B	GPTimer 2A	B	UART5 RX	SB	GPIO C	B	Software	B
7	Software	B	GPTimer 2B	B	UART5 TX	SB	GPIO D	B	Software	B
8	UART0 RX	SB	UART1 RX	SB	Software	B	GPTimer 5A	B	Software	B
9	UART0 TX	SB	UART1 TX	SB	Software	B	GPTimer 5B	B	Software	B
10	SSI0 RX	SB	SSI1 RX	SB	UART6 RX	SB	GPTimer 6A	B	Software	B
11	SSI0 TX	SB	SSI1 TX	SB	UART6 TX	SB	GPTimer 6B	B	Software	B
12	Software	B	UART2 RX	SB	SSI2 RX	SB	GPTimer 7A	B	Software	B
13	Software	B	UART2 TX	SB	SSI2 TX	SB	GPTimer 7B	B	Software	B
14	ADC0 SS0	B	GPTimer 2A	B	SSI3 RX	SB	GPIO E	B	Software	B
15	ADC0 SS1	B	GPTimer 2B	B	SSI3 TX	SB	GPIO F	B	Software	B
16	ADC0 SS2	B	Software	B	UART3 RX	SB	GPTimer 8A	B	Software	B
17	ADC0 SS3	B	Software	B	UART3 TX	SB	GPTimer 8B	B	Software	B
18	GPTimer 0A	B	GPTimer 1A	B	UART4 RX	SB	GPIO B	B	Software	B
19	GPTimer 0B	B	GPTimer 1B	B	UART4 TX	SB	Software	B	Software	B
20	GPTimer 1A	B	Software	B	UART7 RX	SB	Software	B	Software	B
21	GPTimer 1B	B	Software	B	UART7 TX	SB	Software	B	Software	B
22	UART1 RX	SB	Software	B	Software	B	Software	B	Software	B
23	UART1 TX	SB	Software	B	Software	B	Software	B	Software	B
24	SSI1 RX	SB	ADC1 SS0	B	Software	B	GPTimer 9A	B	Software	B
25	SSI1 TX	SB	ADC1 SS1	B	Software	B	GPTimer 9B	B	Software	B
26	Software	B	ADC1 SS2	B	Software	B	GPTimer 10A	B	Software	B
27	Software	B	ADC1 SS3	B	Software	B	GPTimer 10B	B	Software	B
28	Software	B	Software	B	Software	B	GPTimer 11A	B	Software	B
29	Software	B	Software	B	Software	B	GPTimer 11B	B	Software	B
30	Software	B	Software	B	Software	B	Software	B	Software	B
31	Reserved	B	Reserved	B	Reserved	B	Reserved	B	Reserved	B

S = Single
B = Burst
SB = Both

Configuration...

Channel Configuration



Channel Configuration

- Channel control is done via a set of control structures in a table
- The table must be located on a 1024-byte boundary
- Each channel can have one or two control structures; a primary and an alternate
- The primary structure is for BASIC and AUTO transfers. Alternate is for Ping-Pong and Scatter-gather

Control Structure Memory Map

Offset	Channel
0x0	0, Primary
0x10	1, Primary
...	...
0x1F0	31, Primary
0x200	0, Alternate
0x210	1, Alternate
...	...
0x3F0	31, Alternate

Channel Control Structure

Offset	Description
0x000	Source End Pointer
0x004	Destination End Pointer
0x008	Control Word
0x00C	Unused

Control word contains:

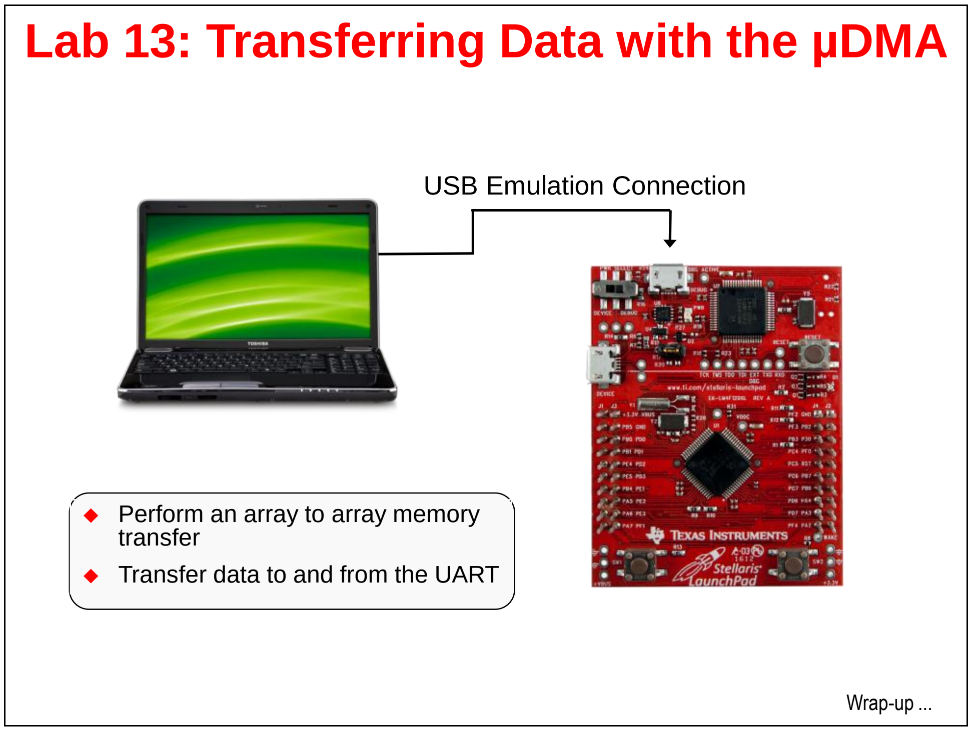
- Source and Dest data sizes
- Source and Dest addr increment size
- # of transfers before bus arbitration
- Total elements to transfer
- Useburst flag
- Transfer mode

Lab...

Lab 13: µDMA

Objective

In this lab you will experiment with the µDMA, transferring arrays of data in memory and then transferring data to and from the UART.



Lab 13: Transferring Data with the µDMA

USB Emulation Connection

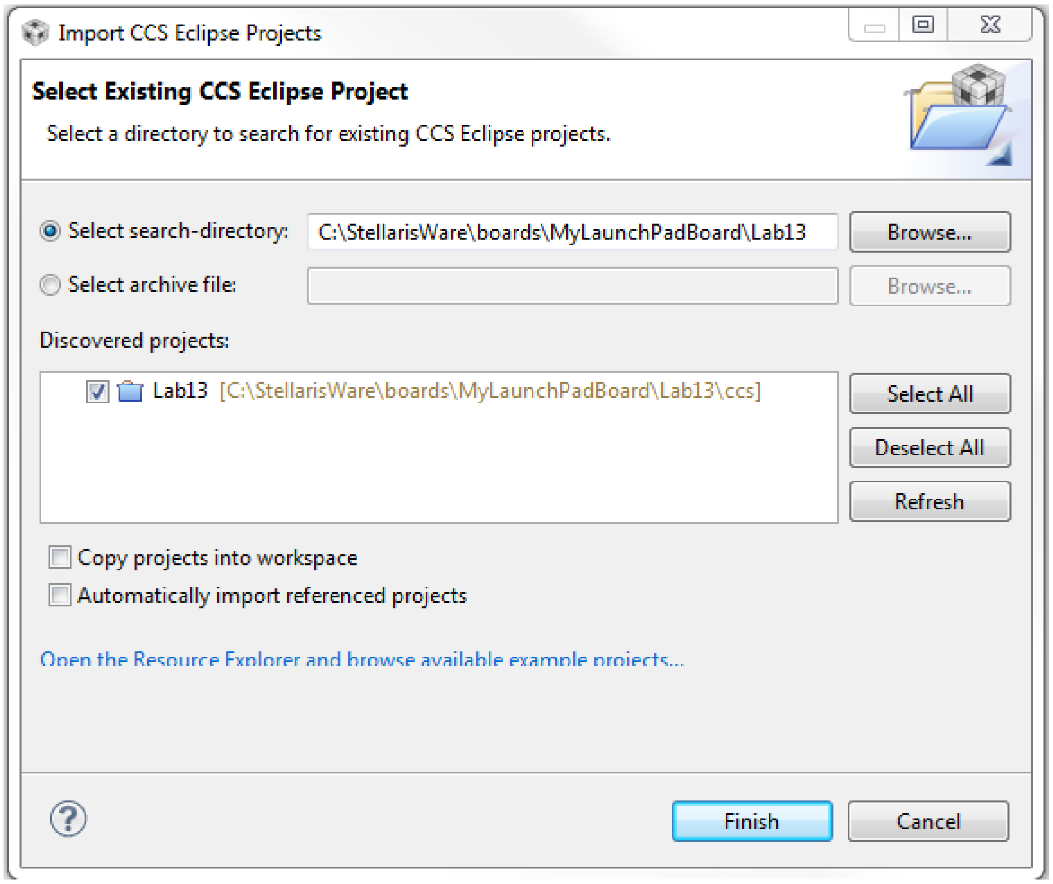
- Perform an array to array memory transfer
- Transfer data to and from the UART

Wrap-up ...

Procedure

Import Lab13

1. We have already created the Lab13 project for you with `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Make sure that the “Copy projects into workspace” checkbox is **unchecked**.



Import CCS Eclipse Projects

Select Existing CCS Eclipse Project

Select a directory to search for existing CCS Eclipse projects.

● Select search-directory: C:\StellarisWare\boards\MyLaunchPadBoard\Lab13

○ Select archive file:

Discovered projects:

☑ Lab13 [C:\StellarisWare\boards\MyLaunchPadBoard\Lab13\ccs]

☐ Copy projects into workspace

☐ Automatically import referenced projects

Open the Resource Explorer and browse available example projects...

Browse the Code

2. In order to save some time, we're going to browse this existing code rather than enter it line by line. Open `main.c` in the editor pane and we'll get started. If you accidentally make a change, this code is also in `main1.txt` in the `Lab13\ccs` folder.

   This code is actually a stripped-down version of the uDMA_demo example in `C:\StellarisWare\boards\ek-lm4f120xl` . To make things a little simpler the UART portion of the code was removed.

   At the top of the code you'll find all the normal includes, especially `udma.h` since we'll be using that peripheral.

3. Just under includes are the definitions for the source and destination buffers, two error counter variables and a counter to track the number of transfers.

```
#define MEM_BUFFER_SIZE         1024
static unsigned long g_ulSrcBuf[MEM_BUFFER_SIZE];
static unsigned long g_ulDstBuf[MEM_BUFFER_SIZE];

static unsigned long g_uluDMAErrCount = 0;
static unsigned long g_ulBadISR = 0;


static unsigned long g_ulMemXferCount = 0;
```

4. Below that, the µDMA control table is defined. Remember that the table must be aligned to a 1024-byte boundary. The #pragma will do that for us. If you are using a different IDE, this construct may be different. The table probably doesn't need to be 1K in length, but that's fine for this example.

```
#pragma DATA_ALIGN(ucControlTable, 1024)
unsigned char ucControlTable[1024];
```

5. Below the control table definition is the library error handler that we've covered earlier.

   Next is the µDMA error handler code. If the µDMA controller encounters a bus or memory protection error as it attempts to perform a data transfer, it disables the µDMA channel that caused the error and generates an interrupt on the µDMA error interrupt vector. The handler here will clear the error and increment the error count.

```
void uDMAErrorHandler(void)
{
    unsigned long ulStatus;
    ulStatus = ROM_uDMAErrorStatusGet();

    if(ulStatus)
    {
        ROM_uDMAErrorStatusClear();
        g_uluDMAErrCount++;
    }
}
```

6. Below the error handler is the µDMA interrupt handler. The interrupt that runs this handler is triggered by the completion of the programmed transfer. The code first checks to see if the µDMA channel is in stop mode. If it is, the transfer count is incremented, the µDMA is set up for another transfer and the next transfer is triggered. If this interrupt was triggered in error, the bad ISR variable will be incremented.

   The last two lines inside the `if()` trigger the second and every subsequent µDMA request.

```
void
uDMAIntHandler(void)
{
    unsigned long ulMode;

    ulMode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SW);
    if(ulMode == UDMA_MODE_STOP)
    {
        g_ulMemXferCount++;

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SW, UDMA_MODE_AUTO,
                                   g_ulSrcBuf, g_ulDstBuf, MEM_BUFFER_SIZE);

        ROM_uDMAChannelEnable(UDMA_CHANNEL_SW);
        ROM_uDMAChannelRequest(UDMA_CHANNEL_SW);
    }
    else
    {
        g_ulBadISR++;
    }
}
```

7. Next is the `InitSWTransfer()` function. This code initializes the µDMA software channel to perform a memory to memory transfer. We'll be triggering these transfers from software, so we'll use the software µDMA channel `(UDMA_CHANNEL_SW).`

   The `for()` construct at the top initializes the source array with a simple pattern.

   The next line enables the µDMA interrupt to the NVIC.

   The next line disables the listed attributes of the software µDMA channel so that it's in a known state.

   The `ROM_uDMAChannelControlSet()` API sets up the control parameters for the software channel µDMA control structure. Notice that we'll be using the primary (not the alternate set) and that the element size and increment sizes are 32-bits. The arbitration count is 8.

   The `ROM_uDMAChannelTransferSet()` API sets up the transfer parameters for the software channel µDMA control structure. Again, this is for the primary set, auto mode (continue transfer until completion even if request is removed … common for software requests), the source and destination buffer addresses and the size of the transfer.

   Finally, the code enables the software channel and makes the first µDMA request.

```
void
InitSWTransfer(void)
{
    unsigned int uIdx;

    for(uIdx = 0; uIdx < MEM_BUFFER_SIZE; uIdx++)
    {
        g_ulSrcBuf[uIdx] = uIdx;
    }

    ROM_IntEnable(INT_UDMA);

    ROM_uDMAChannelControlSetAttributeDisable(UDMA_CHANNEL_SW,
                                    UDMA_ATTR_USEBURST | UDMA_ATTR_ALTSELECT |
                                    (UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK));

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                              UDMA_SIZE_32 | UDMA_SRC_INC_32 | UDMA_DST_INC_32 |
                              UDMA_ARB_8);

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                               UDMA_MODE_AUTO, g_ulSrcBuf, g_ulDstBuf,
                               MEM_BUFFER_SIZE);

    ROM_uDMAChannelEnable(UDMA_CHANNEL_SW);
    ROM_uDMAChannelRequest(UDMA_CHANNEL_SW);
}
```

8. Lastly, we'll look at the code in `main()`.

   - Lazy stacking allows floating point to be used inside interrupt handlers, but uses additional stack space. This isn't strictly needed since we aren't doing any floating-point operations in the handler.
   - Set up the clock to 50MHz.
   - Enable the µDMA peripheral.
   - `ROM_SysCtlPeripheralSleepEnable()` enables the clock to reach this peripheral while the CPU is sleeping. This isn't strictly required here, but if you forget it and put the CPU to sleep, it will be horrible to track down the problem.
   - Then enable the µDMA error interrupt and then the µDMA itself.
   - Make sure the control channel base address is set to the one we created.
   - Call the `InitSWTransfer()` function and start the first transfer, then have the CPU wait in the while(1) loop. In your actual code this would be where you'd either sleep or do something else with those CPU cycles.

```
int
main(void)
{
    ROM_FPULazyStackingEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

    ROM_SysCtlPeripheralClockGating(true);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UDMA);

    ROM_IntEnable(INT_UDMAERR);
    ROM_uDMAEnable();

    ROM_uDMAControlBaseSet(ucControlTable);
    InitSWTransfer();

    while(1)
    {

    }
}
```

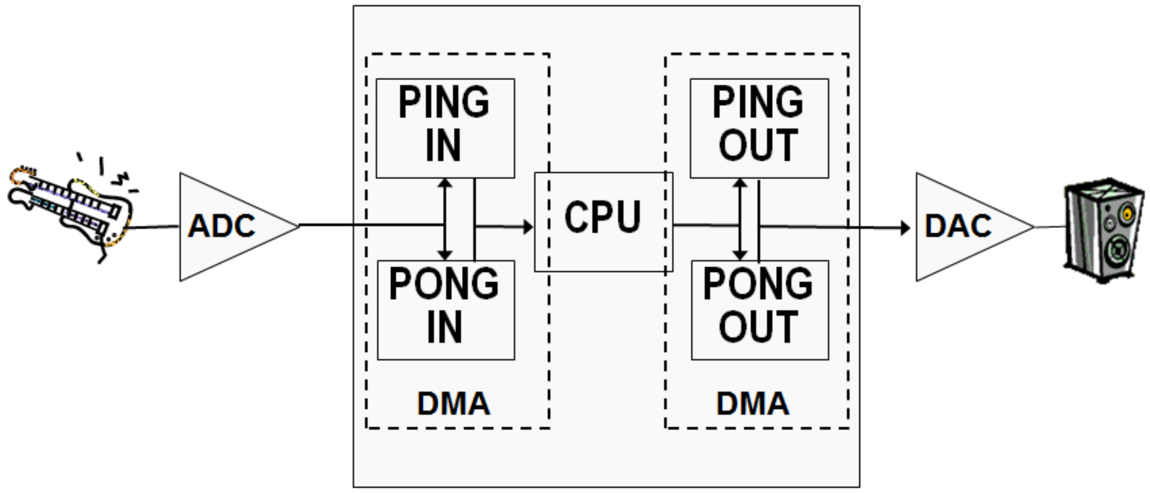
Build, Download and Run the Code

9. Click the Debug button to build and download the code to the LM4F120H5QR flash memory.
10. On the CCS menu bar click View → Memory Browser. Move/resize the window if you have to. Enter `g_ulSrcBuf` in the box below the Memory Browser tab and click the Go button. Click the New Tab button, enter `g_ulDstBuf` in the box and click Go again. Note that both arrays are clear. Click on the `g_ulSrcBuf` tab to view the source array.
11. Set a breakpoint inside the `InitSWTransfer()` function on the line containing `ROM_IntEnable(INT_UDMA);` (about line 98). This will let us see the contents of the source array before any transfers begin.
12. Click the Resume button to run to the breakpoint. In the Memory Browser, note the initialized values in the source array. Check the destination array to make sure it's still clear.
13. Remove the breakpoint you just set and set another one inside the `uDMAIntHandler` function on the line containing `ROM_uDMAChannelTransferSet()`. This breakpoint will occur after the transfer is completed and the transfer count has been incremented, but before the next transfer has been initiated.
14. Add a watch expression on `g_ulMemXferCount`, switch the Memory Browser to the destination tab and click the Resume button. You'll see the destination buffer update with the previous contents of the source buffer and the transfer count variable will now be 1.

    You can click Resume a few times and watch the transfer count increment, but since the source buffer never changes, the destination buffer will look the same after each transfer.
15. Delete the breakpoint you just added. Add watch expressions on `g_ulBadIsr` and `g_uluDMAErrCount`. Click Resume. After a few moments, click the Suspend button. We saw over 200,000 transfers and 0 errors.
16. Remove all of the watch expressions by right-clicking in the Expressions pane and selecting Remove All → Yes. Close the Memory Browser pane.
17. Click the Terminate button to return to the CCS Edit perspective.

Streaming Data To and From the UART using a Ping-Pong Buffer

In real-world applications, incoming or outgoing data doesn't usually stop. If you are receiving data from an ADC or sending/receiving data to/from a UART, the best way to make sure the data always has a place to go to or from is to use a Ping-Pong buffer. Take a filtering application like the one shown below:



Here the DMA on the left is responsible for bringing data from the ADC into memory. When the PING IN buffer is full, the DMA signals the CPU (with an interrupt) and switches its destination to the PONG IN buffer (and vice versa). The CPU filters the frame of data from the PING IN buffer, sends the result to the PING OUT buffer and triggers the DMA on the right to send it to the DAC (and vice versa). This is a straight-forward Input – Process – Output technique. When properly synchronized and timed, all three processes happen simultaneously and there is no chance for a "skip" or "miss" of even a single bit a data, as long as the CPU is capable of processing the buffer of data in the same amount of time that it takes to fill or empty the buffer from/to the outside world.

This example will be a little simpler. We'll have a single transmit buffer, since the data in it won't change. The transmit DMA will send that buffer to the UART transmit register continuously. The UART will be configured in loopback mode so that data will be streaming back in continuously. The receive DMA will stream the data received from the UART data receive register into a Ping-Pong buffer that we can observe.

What makes this DMA programming interesting is that the primary and alternate modes must be used in order for the DMA to switch Ping-Pong buffers automatically. Also, the DMA transfers that point to the UART must not increment, otherwise they would write data into the wrong location. At the same time, the DMA must increment through the Ping and Pong buffer to fill them.

18. Delete all the code in `main.c`. Double-click on `main2.txt` in your Project Explorer pane to open it for editing. Copy the contents of `main2.txt` into your now empty `main.c`. Close `main2.txt` and save your work.
19. In order for this code to build and run, we'll need to make a couple of changes to the interrupt vectors used in `startup_ccs.c` . Open `startup_ccs.c` for editing.
20. It's very easy to make an error in the next three steps. Cut and paste if you can. Find the two lines near the top of the file shown below:

    ```
    extern void uDMAErrorHandler(void);
    extern void uDMAIntHandler(void);
    ```

    and change them to read:

    ```
    extern void uDMAErrorHandler(void);
    extern void UART1IntHandler(void);
    ```

21. Around line 131, find the `uDMAIntHandler` entry for the uDMA Software Transfer vector and change it to `IntDefaultHandler`.
22. Around line 91, find the `IntDefaultHandler` entry for the UART1 Rx and Tx vector and change it to `UART1IntHandler`. Save your work and close `startup_ccs.c`. Follow these last four steps in reverse if you want to go back to the memory to memory transfer example.

Browse the Code

23. Starting at the top, notice the additional includes to handle the UART. Just below them are the definitions for the single Tx and two Rx Ping and Pong buffers. Then you'll find the uDMA error count and transfer count variables.
24. Next is the allocation for the uDMA control table. This table is read by the uDMA peripheral hardware and must be aligned on a 1024-byte boundary.
25. Below the table allocation is the familiar library error routine and the same uDMA error handler from the first part of this lab.

26. The heart of this code is the UART interrupt handler. This ISR is run when the receive ping (primary) or pong (alternate) buffer is full or when the transmit buffer is empty. Note the `ulMode =` lines that determine which event triggered the interrupt.

    In the receive buffers the mode is verified to be stopped and the proper transfer count is incremented. You'll see in the initialization that both the primary and alternate parameters are already set up. When the Ping side of the transfer causes an interrupt, the uDMA is already processing the Pong side, so the `TransferSet` API resets the parameters for the flowing Ping transfer. Note that the source is the UART data register.

    The transmit transfer is a basic transfer and needs to be re-enabled each time it completes. Note that the destination is the same UART data register.

```
void
UART1IntHandler(void)
{
    unsigned long ulStatus;
    unsigned long ulMode;

    ulStatus = ROM_UARTIntStatus(UART1_BASE, 1);

    ROM_UARTIntClear(UART1_BASE, ulStatus);

    ulMode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT);

    if(ulMode == UDMA_MODE_STOP)
    {
        g_ulRxPingCount++;

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                                   UDMA_MODE_PINGPONG,
                                   (void *)(UART1_BASE + UART_O_DR),
                                   g_ucRxPing, sizeof(g_ucRxPing));
    }

    ulMode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT);

    if(ulMode == UDMA_MODE_STOP)
    {
        g_ulRxPongCount++;

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                                   UDMA_MODE_PINGPONG,
                                   (void *)(UART1_BASE + UART_O_DR),
                                   g_ucRxPong, sizeof(g_ucRxPong));
    }

    if(!ROM_uDMAChannelIsEnabled(UDMA_CHANNEL_UART1TX))
    {

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                                   UDMA_MODE_BASIC, g_ucTxBuf,
                                   (void *)(UART1_BASE + UART_O_DR),
                                   sizeof(g_ucTxBuf));

        ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
    }
}
```

27. The µDMA and UART must be initialized and the next function, `InitUART1Transfer()` does that.

    The `for()` loop at the beginning initializes the transmit buffer with some count data.

    The next two lines enable UART1 and make sure that the clock to the peripheral will still be available even if the CPU is sleeping. This last step isn't strictly needed, but many programs utilizing the DMA do sleep and if you forget this step, if will not be easy to track down.

    The next six lines configure the UART clock, the FIFO utilization, enable it, enable it to use the DMA, set loopback mode and enable the interrupt.

    Next up are the µDMA control and transfer programming steps.

    `ROM_uDMAChannelAttributeDisable()` turns off all the indicated parameters to assure the starting point.

    The next two `ROM_uDMAChannelControlSet()` lines set up the control parameters for the Ping (primary) and Pong (alternate) sets. Note that the transfer element size is 8-bits, the source increment is none (since it should be pointing to the UART data register all the time) and the destination increment is 8-bits.

    The next two `ROM_uDMAChannelTransferSet()` lines program the transfer parameters for both the Ping (primary) and Pong (alternate) sets. Note that the mode is `PINGPONG`, the source is the UART data register and the destination is the appropriate Ping or Pong buffer.

    The next four lines set up the control and transfer parameters for the transmit channel. Note that the destination is the UART data register and the source is the single transmit buffer. The element transfer size is 8-bits, the source increment is 8-bits and the destination increment is none.

    In all of these setting the priority has been left as `HIGH`. It doesn't make sense to prioritize the transmit over the receive or vice versa.

    The final two lines enable both µDMA transfers.

```
void InitUART1Transfer(void)
{
    unsigned int uIdx;

    for(uIdx = 0; uIdx < UART_TXBUF_SIZE; uIdx++)
    {
        g_ucTxBuf[uIdx] = uIdx;
    }

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UART1);

    ROM_UARTConfigSetExpClk(UART1_BASE, ROM_SysCtlClockGet(), 115200,
                            UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                            UART_CONFIG_PAR_NONE);

    ROM_UARTFIFOLevelSet(UART1_BASE, UART_FIFO_TX4_8, UART_FIFO_RX4_8);

    ROM_UARTEnable(UART1_BASE);
    ROM_UARTDMAEnable(UART1_BASE, UART_DMA_RX | UART_DMA_TX);

    HWREG(UART1_BASE + UART_O_CTL) |= UART_CTL_LBE;

    ROM_IntEnable(INT_UART1);

    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_UART1RX,
                                    UDMA_ATTR_ALTSELECT | UDMA_ATTR_USEBURST |
                                    UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK);

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 |
                              UDMA_ARB_4);

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 |
                              UDMA_ARB_4);

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                               UDMA_MODE_PINGPONG,
                               (void *)(UART1_BASE + UART_O_DR),
                               g_ucRxPing, sizeof(g_ucRxPing));

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                               UDMA_MODE_PINGPONG,
                               (void *)(UART1_BASE + UART_O_DR),
                               g_ucRxPong, sizeof(g_ucRxPong));

    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_UART1TX,
                                    UDMA_ATTR_ALTSELECT |
                                    UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK);

    ROM_uDMAChannelAttributeEnable(UDMA_CHANNEL_UART1TX, UDMA_ATTR_USEBURST);


    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE |
                              UDMA_ARB_4);

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                               UDMA_MODE_BASIC, g_ucTxBuf,
                               (void *)(UART1_BASE + UART_O_DR),
                               sizeof(g_ucTxBuf));

    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1RX);
    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);

}
```

28. Finally we're in main().

    Starting at the top we have the lazy stacking enable, which probably isn't necessary since we're not using the PFU in the handlers.

    The clock is set up to 50MHz and the peripherals are allowed to be clocked during sleep mode.

    GPIO port F is enabled and set up for the LEDs. We'll only be using the blue LED.

    The next five lines set up the hardware for the UART on port A pins 0 and 1.

    The five lines afterwards enable the uDMA clock, allow it to operate during sleep modes, enable the error interrupt, enable the uDMA for operation and sets the base address for the uDMA control table.

    Then the initialization function is called for the transfers.

    The `while(1)` loop simply blinks the blue LED while the transfers are happening to let us know the code is alive.

```
int main(void)
{
    volatile unsigned long ulLoop;


    ROM_FPULazyStackingEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

    ROM_SysCtlPeripheralClockGating(true);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UART0);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UDMA);
    ROM_IntEnable(INT_UDMAERR);
    ROM_uDMAEnable();
    ROM_uDMAControlBaseSet(ucControlTable);

    InitUART1Transfer();


    while(1)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);

        SysCtlDelay(SysCtlClockGet() / 20 / 3);

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);

        SysCtlDelay(SysCtlClockGet() / 20 / 3);
    }
}
```

Build, Load and Run

29. Click the Debug button to build and load the program.

30. In order to determine of the program is operating properly, we need to see the buffers. One the CCS menu bar, click View → Memory Browser. Enter `g_ucRxPing` in the box below the Memory Browser tab and click the Go button. The RxPing, RxPong and Tx buffers are all close together, so you should be able to see them in the same window. Resize if necessary.

31. Notice that the Tx buffer is clear. Set a breakpoint in the `InitUART1Transfer()` function on the line containing `ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);` . This is right after the Tx buffer is initialized with data.

32. Click the Resume button to run to the breakpoint. Note in the Memory Browser that the Tx buffer is now filled with data.

33. Remove the breakpoint and set another in `UART1IntHandler()` on the line containing `ulStatus =` . This breakpoint will trip when the first (Pong) transfer completes

34. Click the Resume button to run to the breakpoint. Note in the Memory Browser that the `RxPing` buffer is now filled with data. Click Resume again and the `RxPong` buffer will fill.

35. Add a watch expressions on `g_ulRxPingCount` and `g_ulRxPingCount` found in `UART1IntHandler()` . Add another watch expression on `g_uluDMAErrCount` found in `uDMAErrorHandler()` . Change the properties of the breakpoint so that its Action is Refresh All Windows.

36. Click Resume. The transfer counters should track and the error count should be zero. You'll also notice that the LED on the LaunchPad stops blinking. Since the CPU is stopping at the breakpoint and transferring data to the PC, the next uDMA interrupt occurs before any code can run in the while(1) loop. Consider that when using this technique to debug.

    The Memory browser isn't very interesting since the Tx buffer never changes. Let's fix that.

37. Halt the code and find the Tx buffer portion of the `UART1IntHandler`. Add the line highlighted below. This will increment the first location in the Tx buffer (and yes, I know that it's cast as a character):

```
if(!ROM_uDMAChannelIsEnabled(UDMA_CHANNEL_UART1TX))
{
    g_ucTxBuf[0]++;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                               UDMA_MODE_BASIC, g_ucTxBuf,
                               (void *)(UART1_BASE + UART_O_DR),
                               sizeof(g_ucTxBuf));

    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
}
```

38. Build, load and Run. You may need to click the Go button in the Memory Browser again. The first location in all three buffers should be incrementing.
39. When you're done, click the Terminate button to return to the CCS Edit perspective. Now that the CCS windows aren't being updated, the blue LED will start blinking again.
40. Right-click on Lab13 in the Project Explorer pane and close the project.
41. Close Code Composer Studio.

You're done.