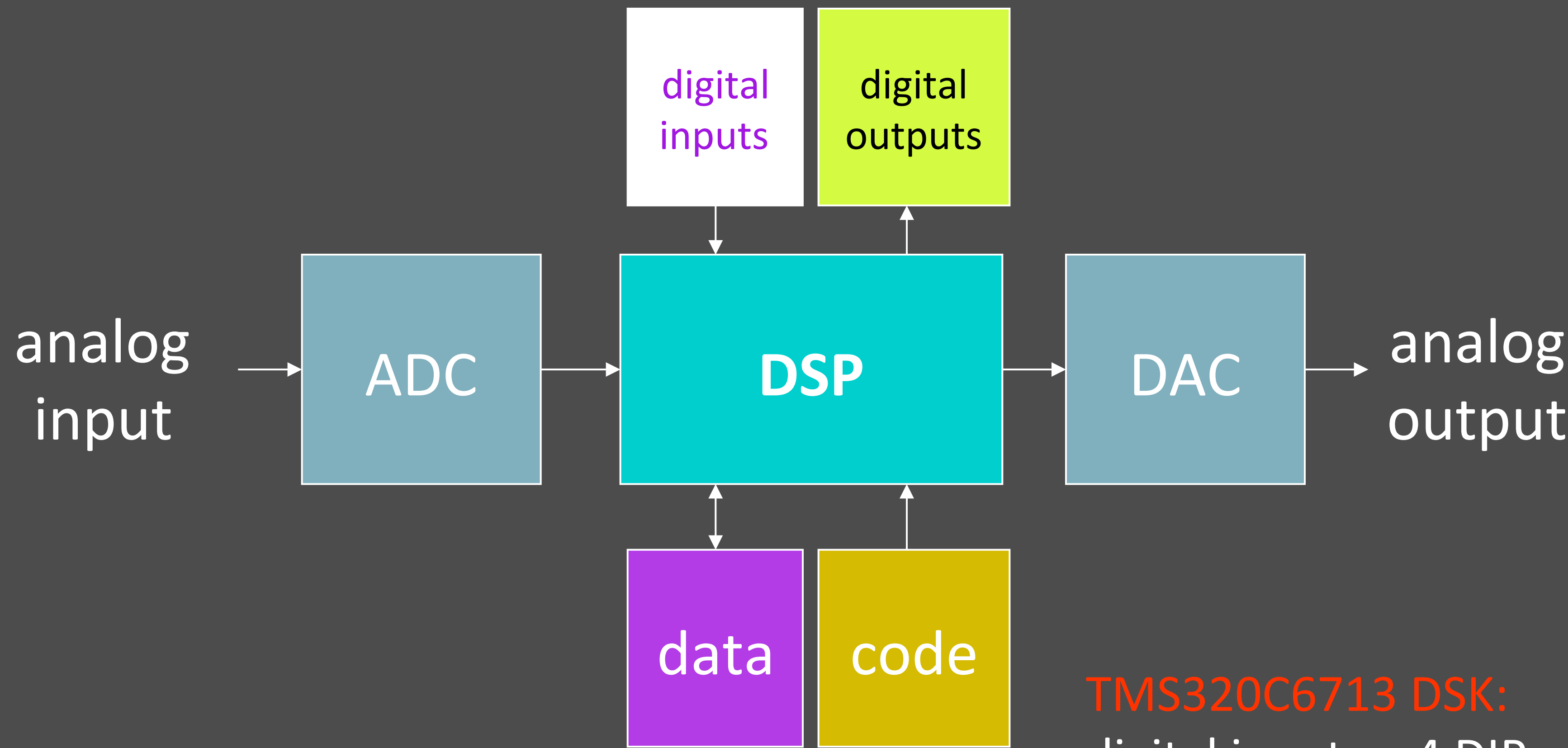
D. Richard Brown III
Associate Professor
Worcester Polytechnic Institute
Electrical and Computer Engineering Department
drb@ece.wpi.edu

27-October-2011

ECE4703 Real-Time DSP: Interfacing with I/O, Debugging, and Profiling

Interfacing a DSP With the Real World



TMS320C6713 DSK:
digital inputs = 4 DIP switches
digital outputs = 4 LEDs
ADC and DAC = AIC23 codec

DIP Switches and LEDs

LED and DIP switch interface functions are provided in dsk6713bsl.lib.

- Initialize the DSK with the BSL function DSK6713_init();
- Initialize DIP/LEDs with
  DSK6713_DIP_init() and/or DSK6713_LED_init()
- Read state of DIP switches with
  DSK6713_DIP_get(n)
- Change state of LEDs with
  DSK6713_LED_on(n) or
  DSK6713_LED_off(n) or
  DSK6713_LED_toggle(n)

where n=0, 1, 2, or 3.

Documentation is available in Board Support Library API (on course website).

AIC23 Codec

- AIC23 codec performs both ADC and DAC functions
- Stereo input and output (left+right channels)
- Initialization steps:
  - Initialize the DSK with the BSL function DSK6713_init();
  - Open the codec with the BSL function
    hCodec = DSK6713_AIC23_openCodec(0,&config);
    - "hCodec" is the codec "handle". You can think of this as a unique address of the codec on the McBSP bus.
    - "config" is the default configuration of the codec. See the header file dsk6713_aic23.h and the AIC23 codec datasheet (link on the course web page) for details.
  - Optional: Set the codec sampling frequency.
  - Configure the McBSP to transmit/receive 32 bits (two 16 bit samples) with the CSL function McBSP_FSETS()
  - Set up and enable interrupts

Codec Initialization Example (from Kehtarnavaz)

Initialization steps:

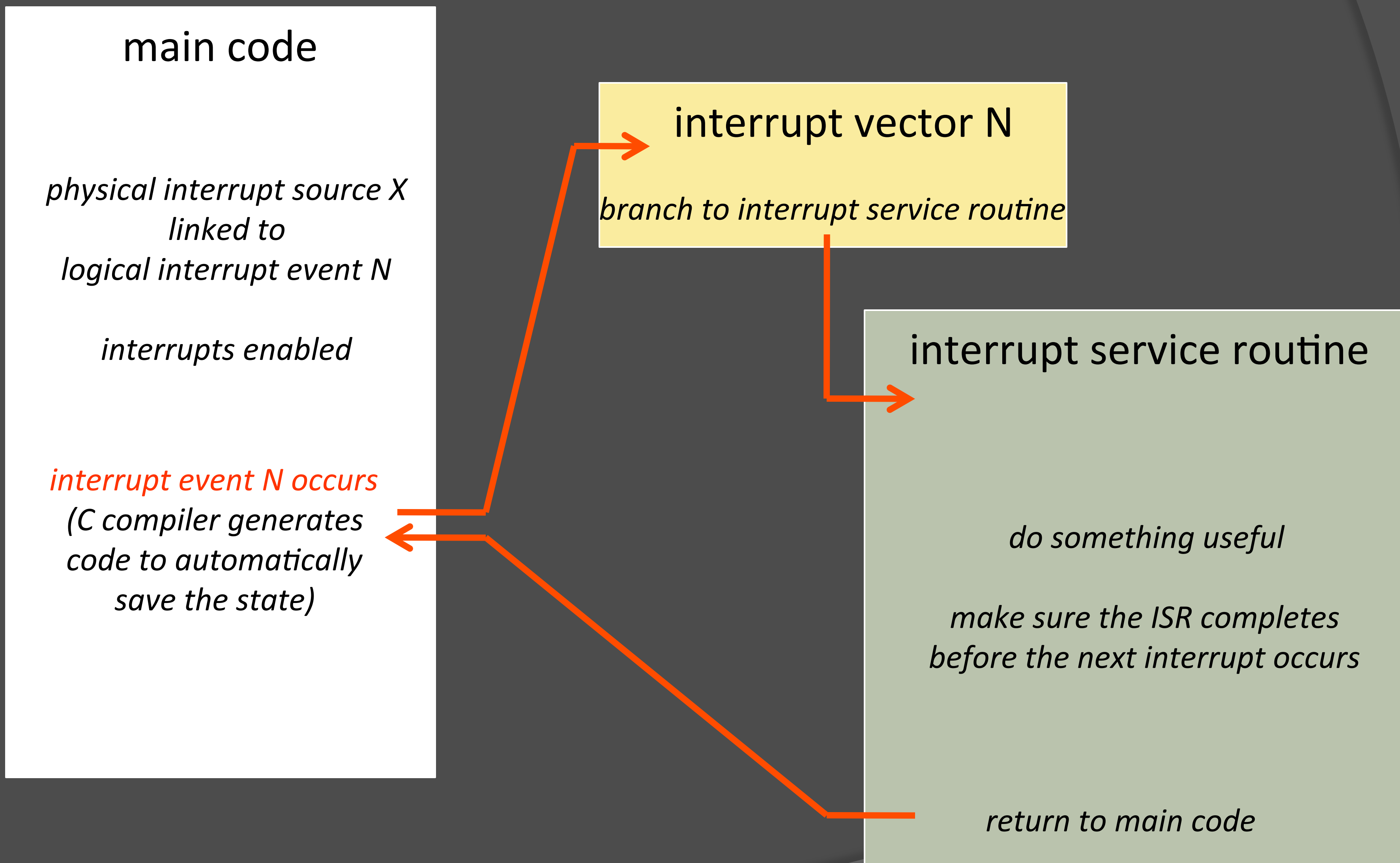
1. Initialize the DSK
2. Open the codec with the default configuration.
3. Configure multi-channel buffered serial port (McBSP)
   - SPCR = serial port control register
   - RCR = receive control register
   - XCR = transmit control register
   - See SPRU508e.pdf
4. Set the sampling rate
5. Configure and enable interrupts
6. Do normal processing (we just enter a loop here)

```c
interrupt void serialPortRcvISR(void);                        // ISR function prototype

void main()
{
    DSK6713_init();     // Initialize the board support library, must be called first
    hCodec = DSK6713_AIC23_openCodec(0, &config);             // Open the codec

    // Configure buffered serial ports for 32 bit operation
    // This allows transfer of both right and left channels in one read/write
    MCBSP_FSETS(SPCR1, RINTM, FRM);
    MCBSP_FSETS(SPCR1, XINTM, FRM);
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);

    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_48KHZ);    // set the sampling rate

    // Interrupt setup
    IRQ_globalDisable();              // Globally disables interrupts
    IRQ_nmiEnable();                  // Enables the NMI interrupt
    IRQ_map(IRQ_EVT_RINT1,15);        // Maps an event to a physical interrupt
    IRQ_enable(IRQ_EVT_RINT1);        // Enables the event
    IRQ_globalEnable();               // Globally enables interrupts

    while(1)
    {
    }
}
```

AIC23 Codec: Interrupts

- We will use an interrupt interface between the DSP and the codec.
- DSP can do useful things while waiting for samples to arrive from codec, e.g. check DIP switches
- C6x interrupt basics:
  - Interrupt sources must be mapped to interrupt events
    - 16 physical "interrupt sources" (timers, serial ports, codec, …)
    - 12 logical "interrupt events" (INT4 to INT15)
  - Interrupt events have associated "interrupt vectors". An "interrupt vector" is a special pointer to the start of the "interrupt service routine" (ISR).
  - Interrupt vectors must be set up in your code (usually in the file "vectors.asm").
  - You are also responsible for writing the ISR.

Interrupts



main code
physical interrupt source X
linked to
logical interrupt event N
interrupts enabled
interrupt event N occurs
(C compiler generates
code to automatically
save the state)
interrupt vector N
branch to interrupt service routine
interrupt service routine
do something useful
make sure the ISR completes
before the next interrupt occurs
return to main code

Interrupt Vector

- We usually link the physical codec interrupt to INT15.
- The ISR in this example is called “serialPortRcvISR” (you can rename it if you like).
- C function “x” is called “_x” in ASM files.
- The interrupt vector is usually in the vectors.asm file:
- Each interrupt vector must be exactly 8 ASM instructions

```
INT15:
    MVKL .S2 _serialPortRcvISR, B0
    MVKH .S2 _serialPortRcvISR, B0
    B    .S2 B0
    NOP
    NOP
    NOP
    NOP
    NOP
```

A Simple Interrupt Service Routine

```
interrupt void serialPortRcvISR()
{
    Uint32 temp;

    temp = MCBSP_read(DSK6713_AIC23_DATAHANDLE);  // read L+R channels
    MCBSP_write(DSK6713_AIC23_DATAHANDLE,temp);   // write L+R channels
}
```

Remarks:

- MCBSP_read() requests L+R samples from the codec’s ADC
- MCBSP_write() sends L+R samples to the codec’s DAC
- This ISR simply reads in samples and then sends them back out.

Setting the Codec Sampling Frequency

Here we open the codec with the default configuration:

```
26    hCodec = DSK6713_AIC23_openCodec(0, &config);              // Open the codec
```

The structure "config" is declared in dsk6713_aic23.h

Rather than editing the default configuration in the header file, we can change the sampling frequency after the initial configuration:

```
35    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_48KHZ);    // set the sampling rate
```

Frequency definitions are in dsk6713_aic.h

```
/* Frequency Definitions */
#define DSK6713_AIC23_FREQ_8KHZ        1
#define DSK6713_AIC23_FREQ_16KHZ       2
#define DSK6713_AIC23_FREQ_24KHZ       3
#define DSK6713_AIC23_FREQ_32KHZ       4
#define DSK6713_AIC23_FREQ_44KHZ       5
#define DSK6713_AIC23_FREQ_48KHZ       6
#define DSK6713_AIC23_FREQ_96KHZ       7
```

This is actually 44.1kHz

Other Codec Configuration

- Line input volume level (individually controllable for left and right channels)
- Headphone output volume level (individually controllable for left and right channels)
- Digital word size (16, 20, 24, or 32 bit)
- Other settings, e.g. byte order, etc. For more details, see:
  - dsk6713_aic23.h
  - AIC23 codec datasheet (link on course web page)

Codec Data Format and How To Separate the Left/Right Channels

```
// we can use the union construct in C to have
// the same memory referenced by two different variables
union {Uint32 combo; short channel[2];} temp;
```


```
// the McBSP functions require that we
// read/write data to/from the Uint32 variable
temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo);

// but if we want to access the left/right channels individually
// we can do this through the short variables
Leftchannel = temp.channel[1];
Rightchannel = temp.channel[0];
```

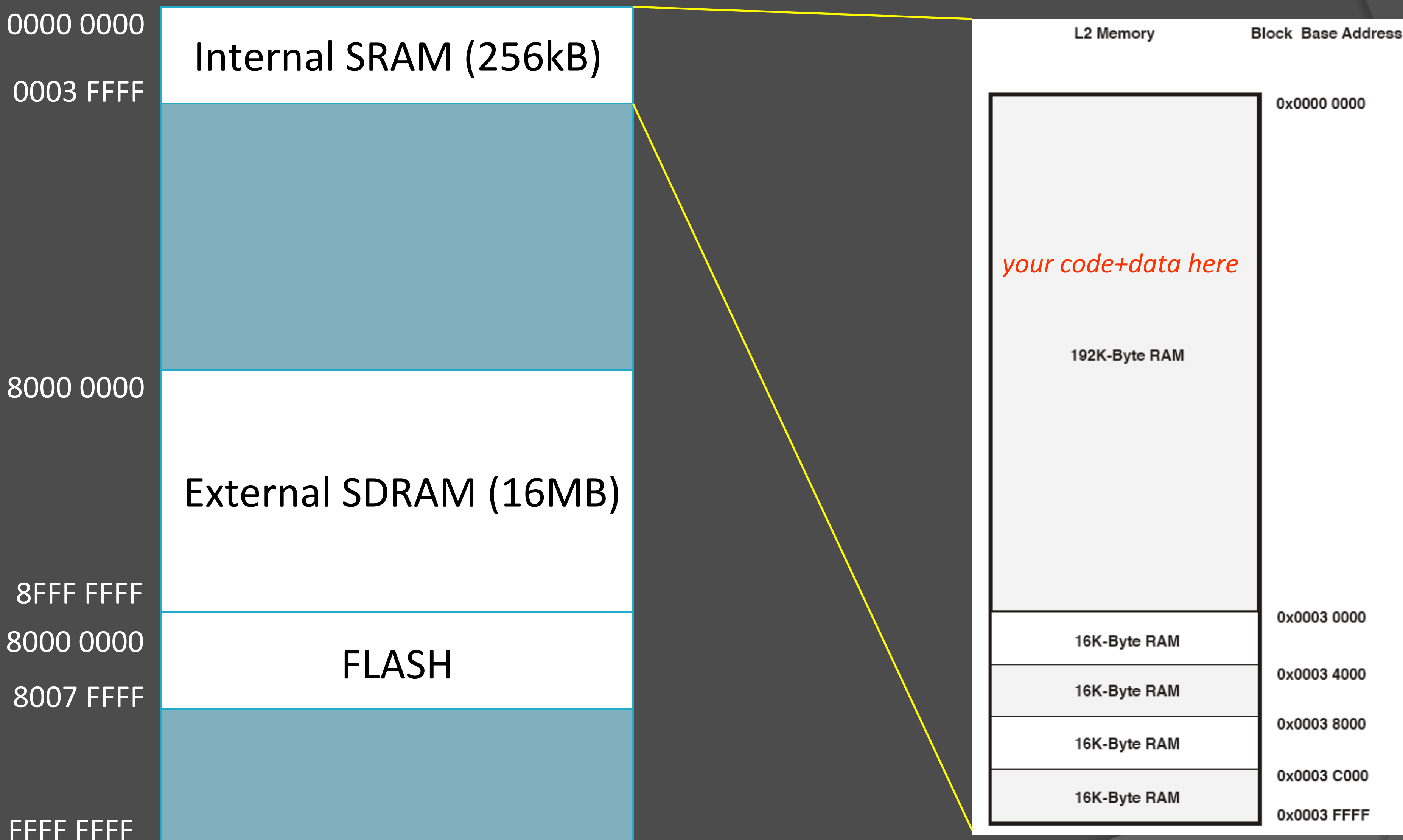
Final Remarks on DSP/Codec Interface

- In most real-time DSP applications, you process samples as they become available from the codec's ADC (sample-by-sample operation).
- This means that all processing will be done in the ISR.
  - **MCBSP_read()**
  - **--- processing here ---**
  - **MCBSP_write()**
- The ISR must run in real-time, i.e. the total execution time must be less than one sampling period.
- You can do DIP/LED processing outside of the ISR (in your main code).

C6713 DSK Memory Architecture

- TSM320C6713 DSP chip has 256kB internal SRAM
  - Up to 64kB of this SRAM can be configured as shared L2 cache
- DSK provides additional 16MB external RAM (SDRAM)
- DSK also provides 512kB external FLASH memory
- Code location (.text in linker command file)
  - internal SRAM memory (fast)
  - external SDRAM memory (typically 2-4x slower, depends on cache configuration)
- Data location (.data in linker command file)
  - internal SRAM memory (fast)
  - external SDRAM memory (slower, depends on datatypes and cache configuration)
- Code+data for all projects assigned in ECE4703 should fit in the C6713 internal SRAM

TMS320C6713 DSK Memory Map



Linker Command File Example

```
MEMORY
{
    vecs:       o = 00000000h    l = 00000200h
    IRAM:       o = 00000200h    l = 0002FE00h
    CE0:        o = 80000000h    l = 01000000h
}

SECTIONS
{
    .vectors    >       vecs
    .cinit      >       IRAM
    .text       >       IRAM
    .stack      >       IRAM
    .bss        >       IRAM
    .const      >       IRAM
    .data       >       IRAM
    .far        >       IRAM
    .switch     >       IRAM
    .sysmem     >       IRAM
    .tables     >       IRAM
    .cio        >       IRAM
}
```

Code goes here

Data goes here

Addresses 00000000-0002FFFF correspond to the lowest 192kB of internal memory (SRAM) and are labeled “IRAM”.

External memory is mapped to address range 80000000 – 80FFFFFF. This is 16MB and is labeled “CEO”.

Both code and data are placed in the C6713 internal SRAM in this example. Interrupt vectors are also in SRAM.

vectors.asm

- This file contains your interrupt vectors
- ".sect" directive at top of file tells linker where (in memory) to put the code
- Each interrupt vector is composed of exactly 8 assembly language instructions
- Example:

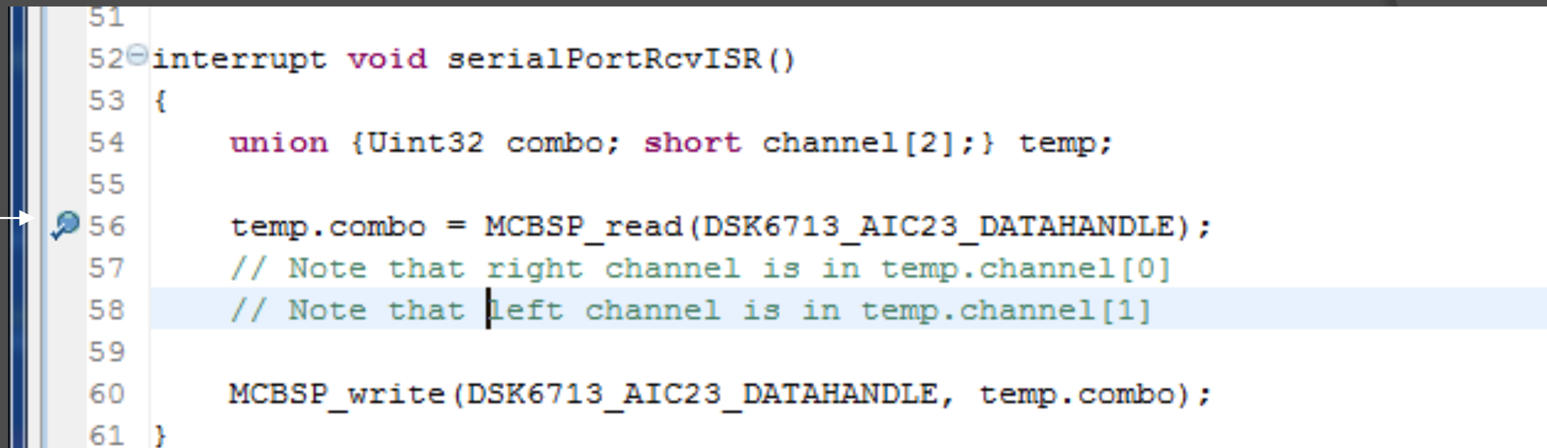
```
INT15:
        MVKL .S2 _serialPortRcvISR, B0
        MVKH .S2 _serialPortRcvISR, B0
        B    .S2 B0
        NOP
        NOP
        NOP
        NOP
        NOP
```

Debugging and Other Useful Features of the CCS IDE

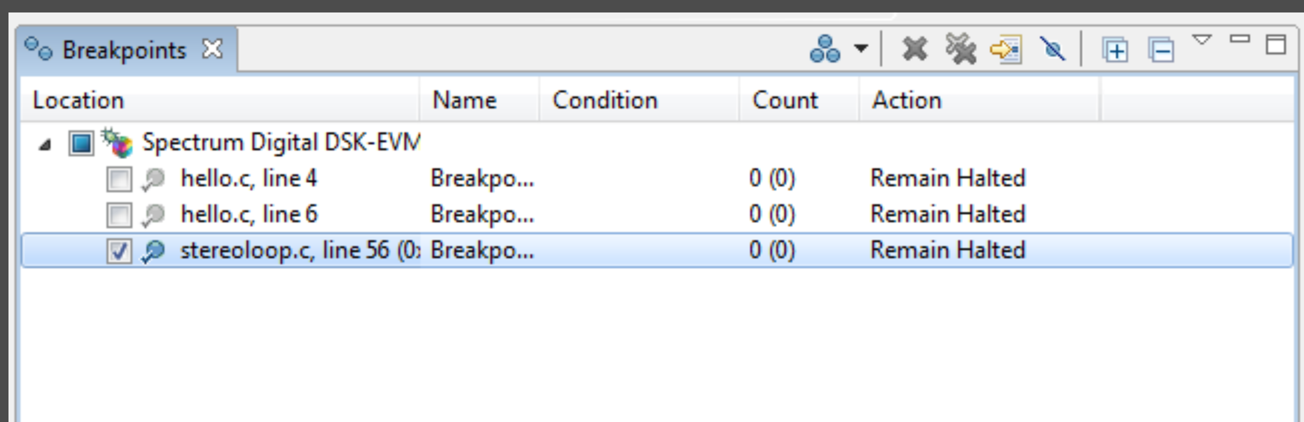
- Breakpoints and stepping through your code
- Watch variables
- Registers
- Plotting arrays of data

Breakpoints: Just Double-Click

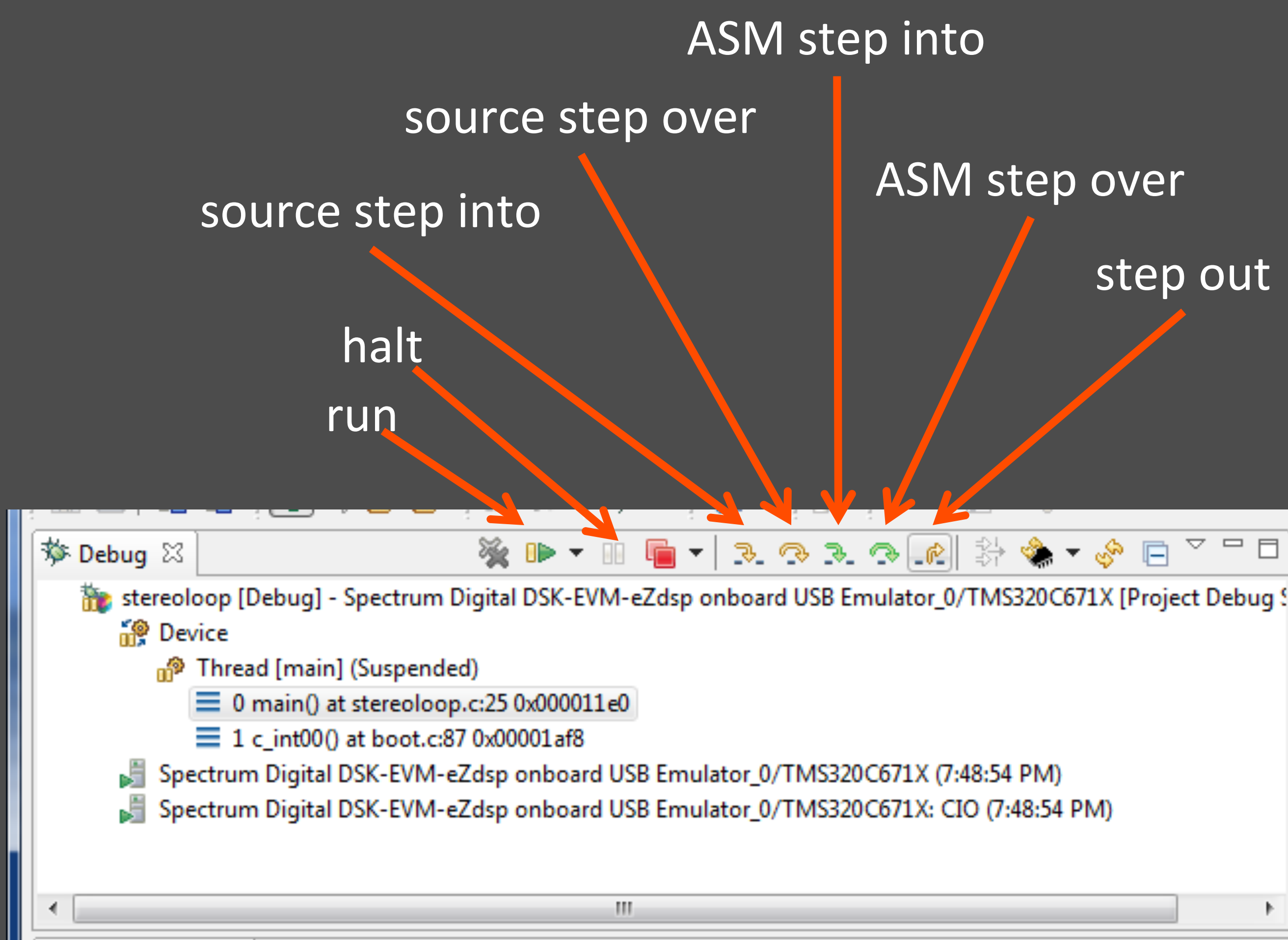
break point



- **Breakpoints:** stop code execution at this point to allow state examination and step-by-step execution.
- Also try View->Breakpoints

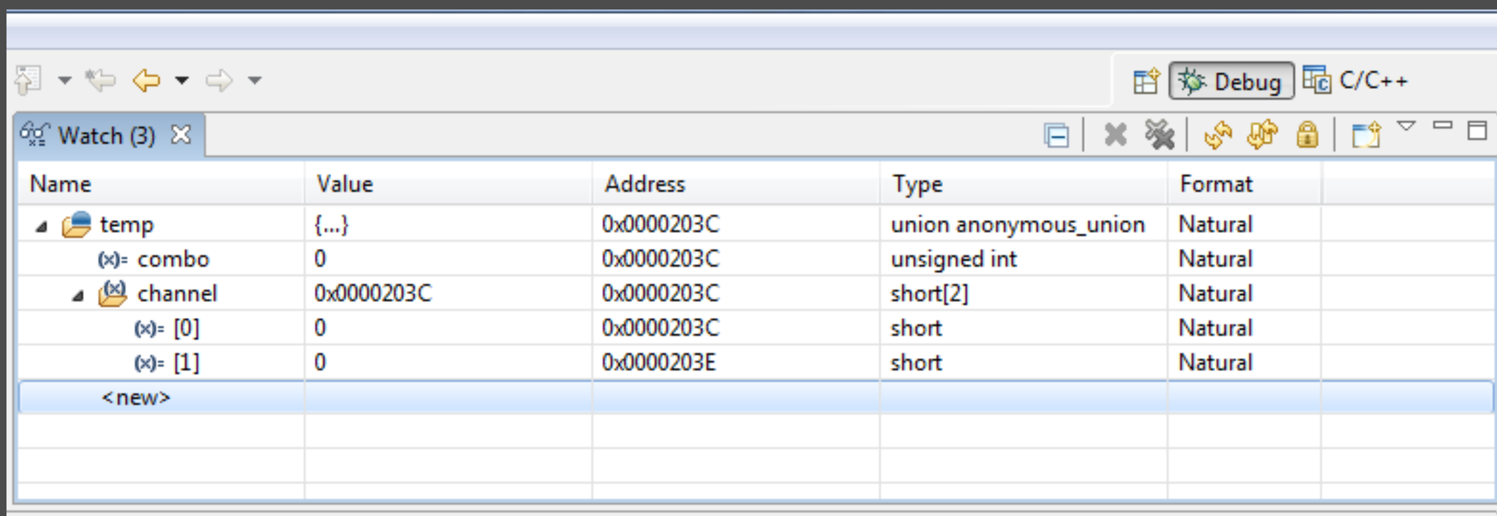


Using Breakpoints



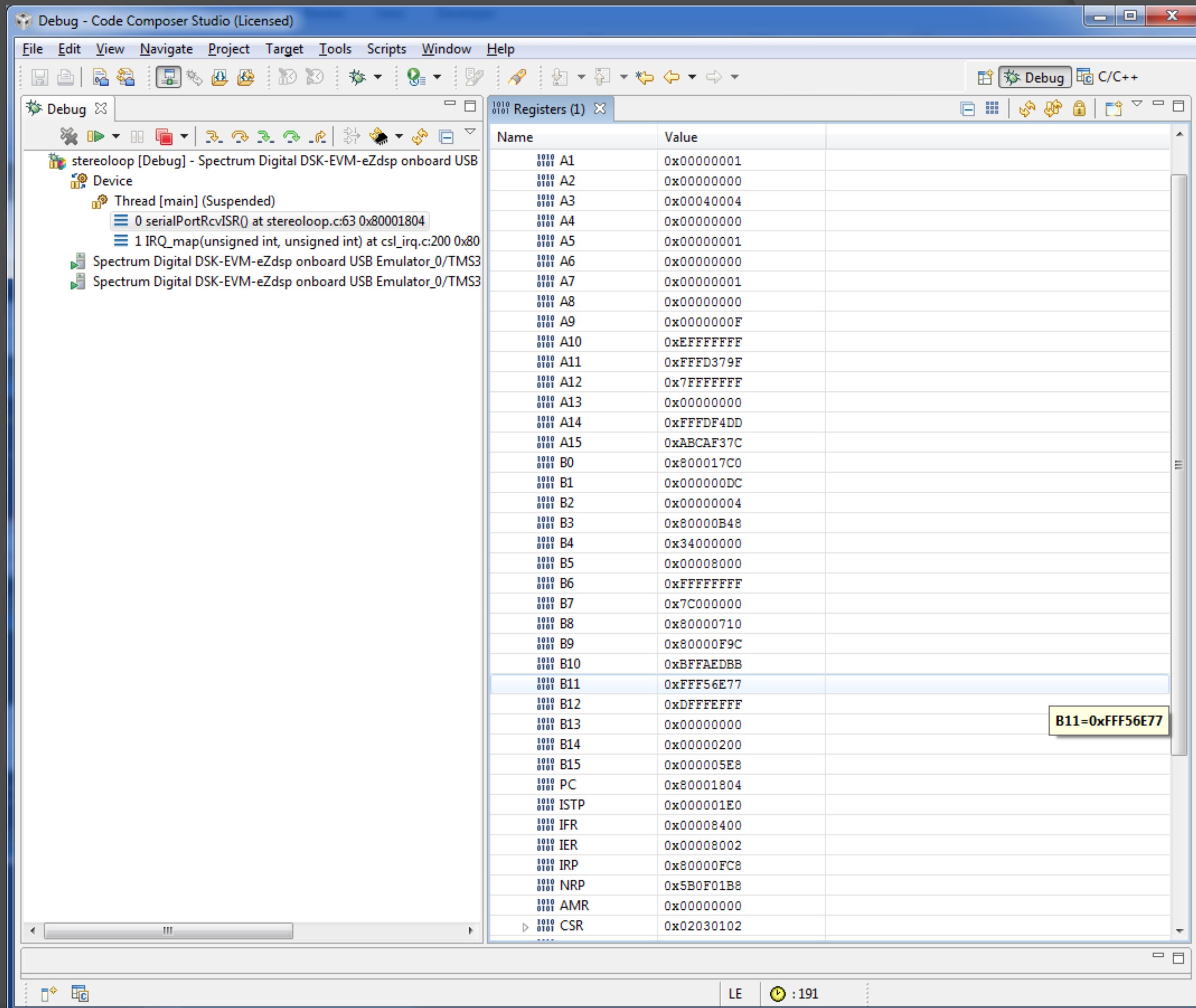
Watch Variables

- View->Watch



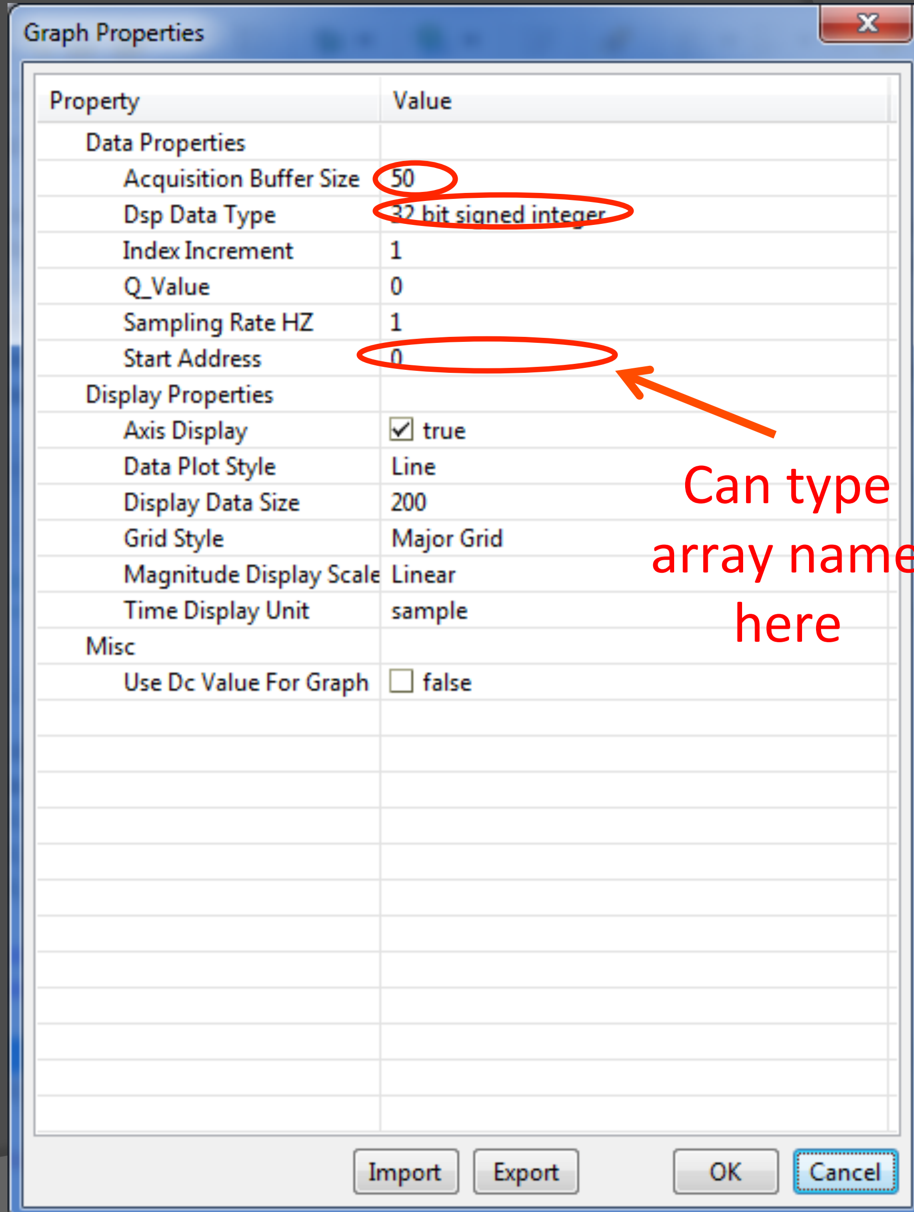
- Type in any variable name currently in scope.
- Tip: Right click on <new> to easily add global variables.

Registers: View->Registers

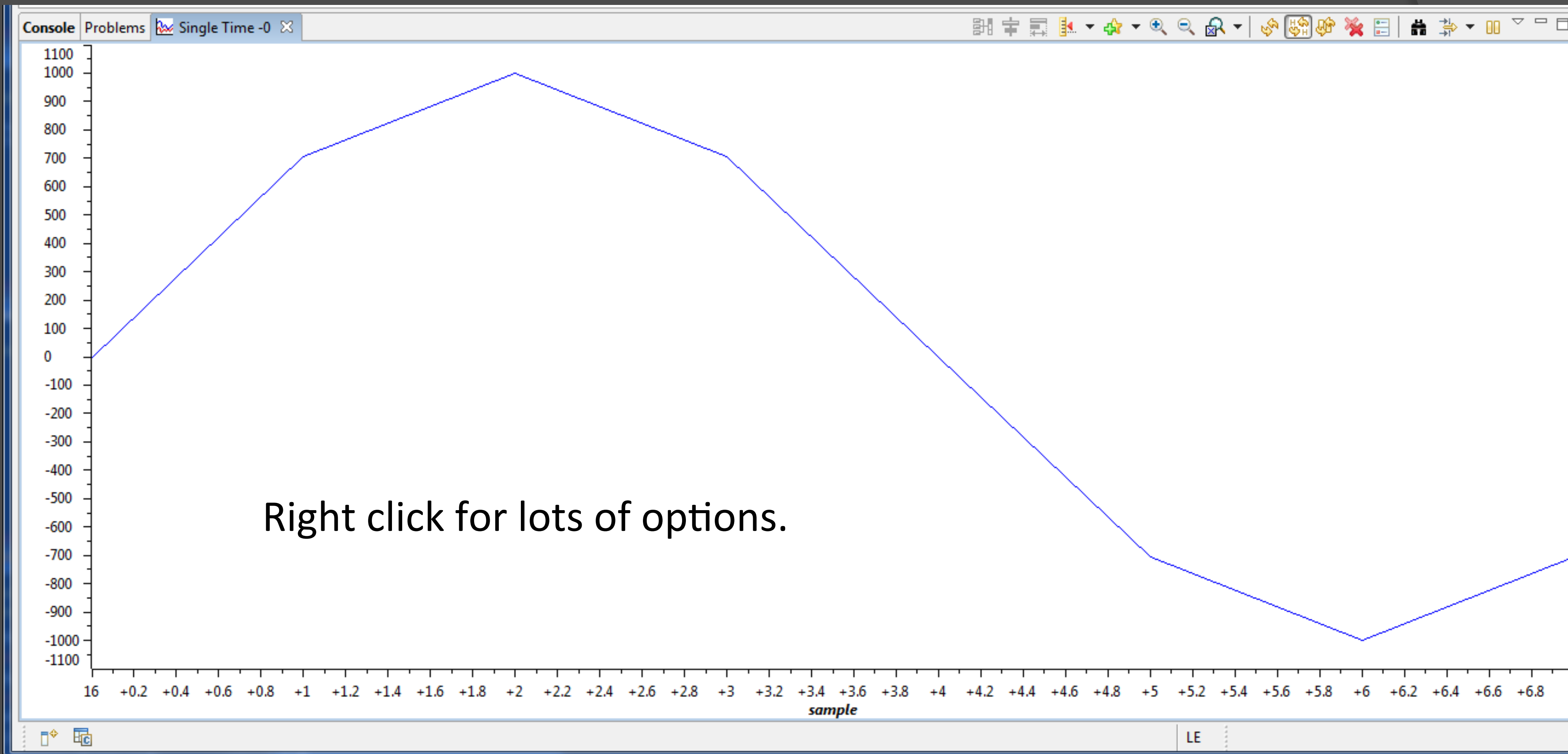


Plotting Arrays of Data

- Tools -> Graph -> (Typically Single Time)



Graph Windows: Plotting Arrays of Data



Profiling Your Code and Making it More Efficient

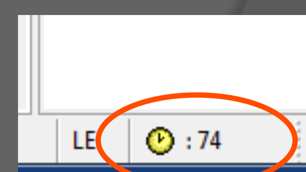
- How to estimate the execution time of your code.
- How to use the optimizing compiler to produce more efficient code.
- Other factors affecting the efficiency of your code.

How to estimate code execution time when connected to the DSK

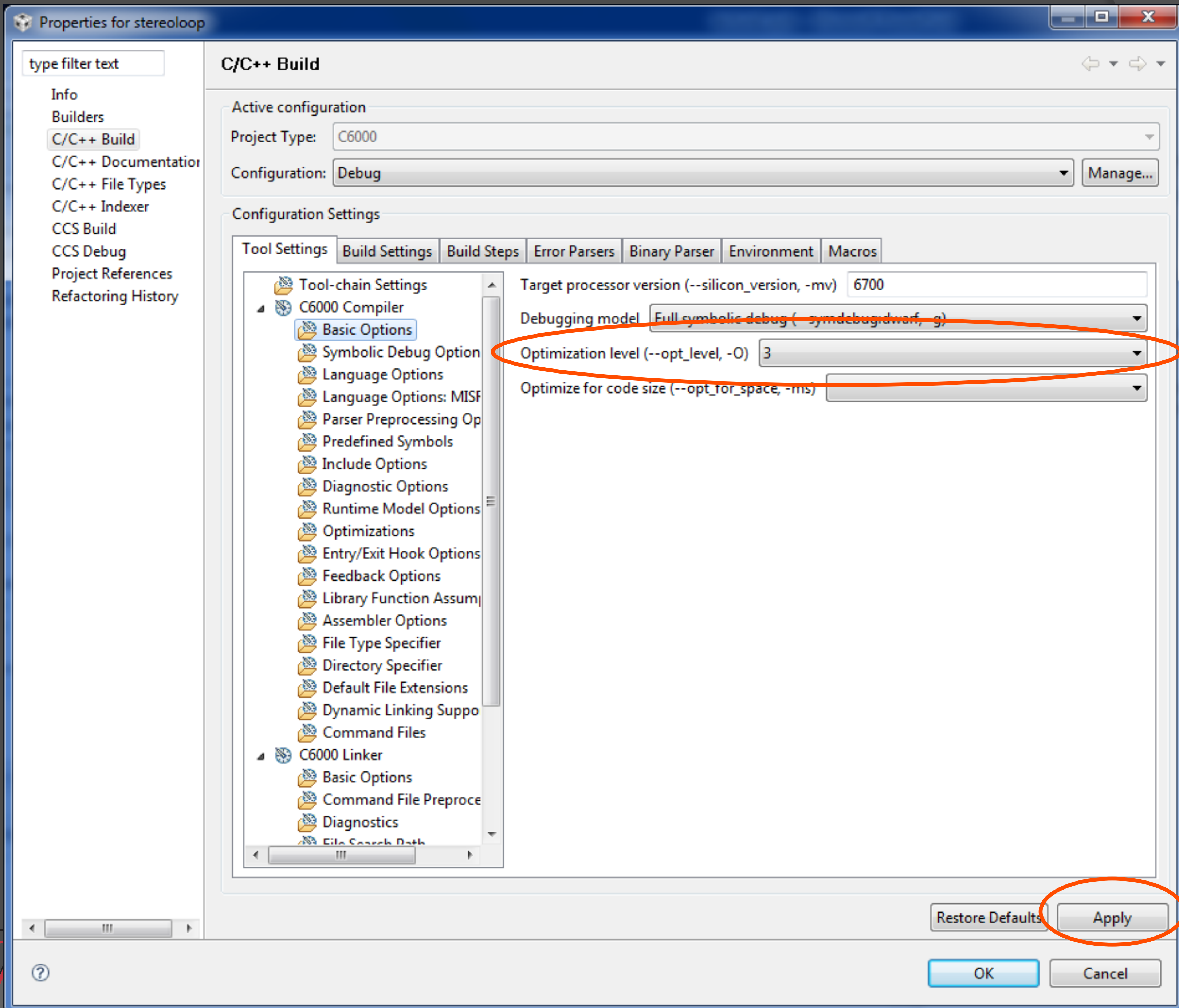
1. Open the source file you wish to profile
2. Set two breakpoints for the start/end of the code range you wish to profile

```
interrupt void serialPortRcvISR()
{
    union {Uint32 combo; short channel[2];} temp;

    temp.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
    // Note that right channel is in temp.channel[0]
    // Note that left channel is in temp.channel[1]

    MCBSP_write(DSK6713_AIC23_DATAHANDLE, temp.combo);
}
```

3. Build it and load .out file to the DSK
4. Target -> Clock -> Enable
5. Target -> Clock -> View
6. Run to the first breakpoint
7. Target -> Clock -> Reset (or double click the clock to reset the clock to zero)
8. Run to the second breakpoint
9. Clock will show raw number of execution cycles between breakpoints.

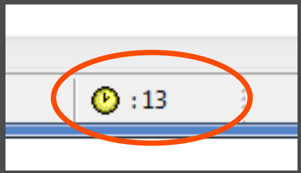


Optimizing Compiler



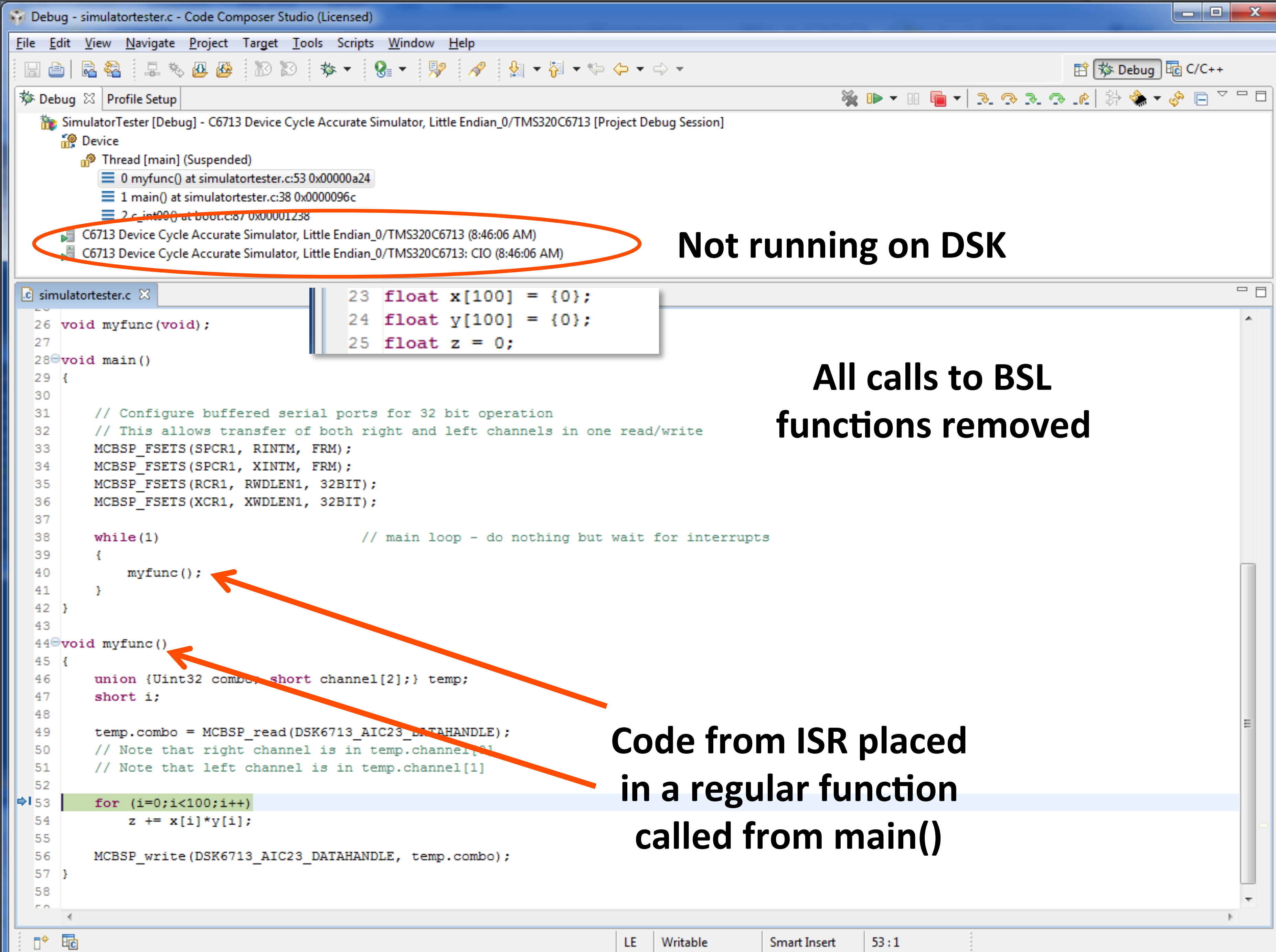
Profiling results after compiler optimization

- Rebuild and reload the program to the DSK
- Use your breakpoint/clock method to profile the execution time
- In this example, we get a 5x-6x improvement with Level-3 Optimization
- Optimization gains can be much larger, e.g. 20x



Limitations of hardware profiling

- Variability of results
- Profiling is known to be somewhat inaccurate when connected to real hardware
- Breakpoint/clock profiling method may not always work with compiler-optimized code
- For the best results, TI recommends profiling your code in a cycle accurate simulator:
  - New target configuration:
    - Connection = Texas Instruments Simulator
    - Device = C6713 Device Cycle Accurate Simulator, Little Endian
  - Need to create a new project for the simulator and copy your functions/code for profiling to this project without calls to board-specific functions
  - Tools -> Profile -> Setup and then Tools-> Profile -> View

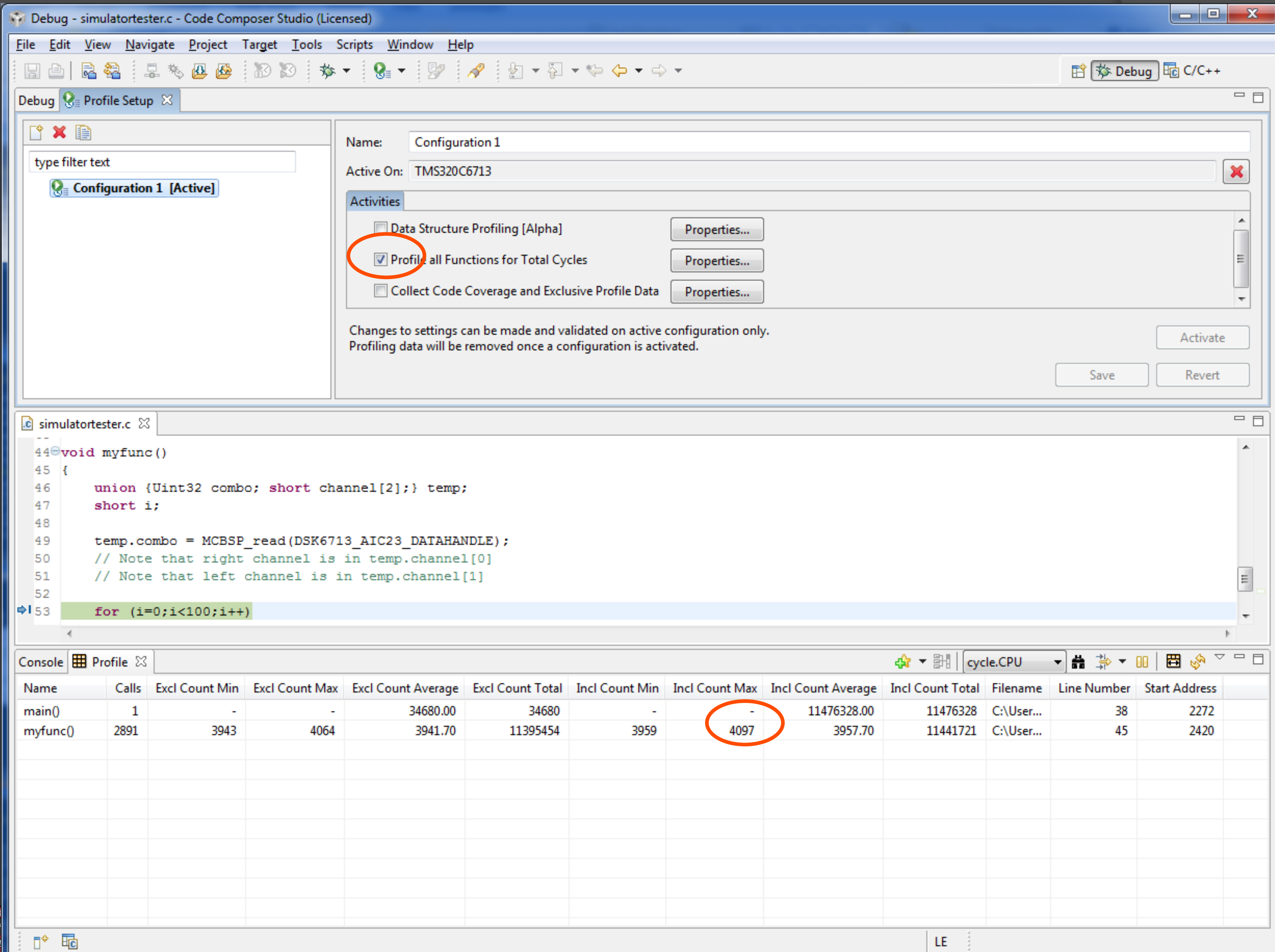


Not running on DSK

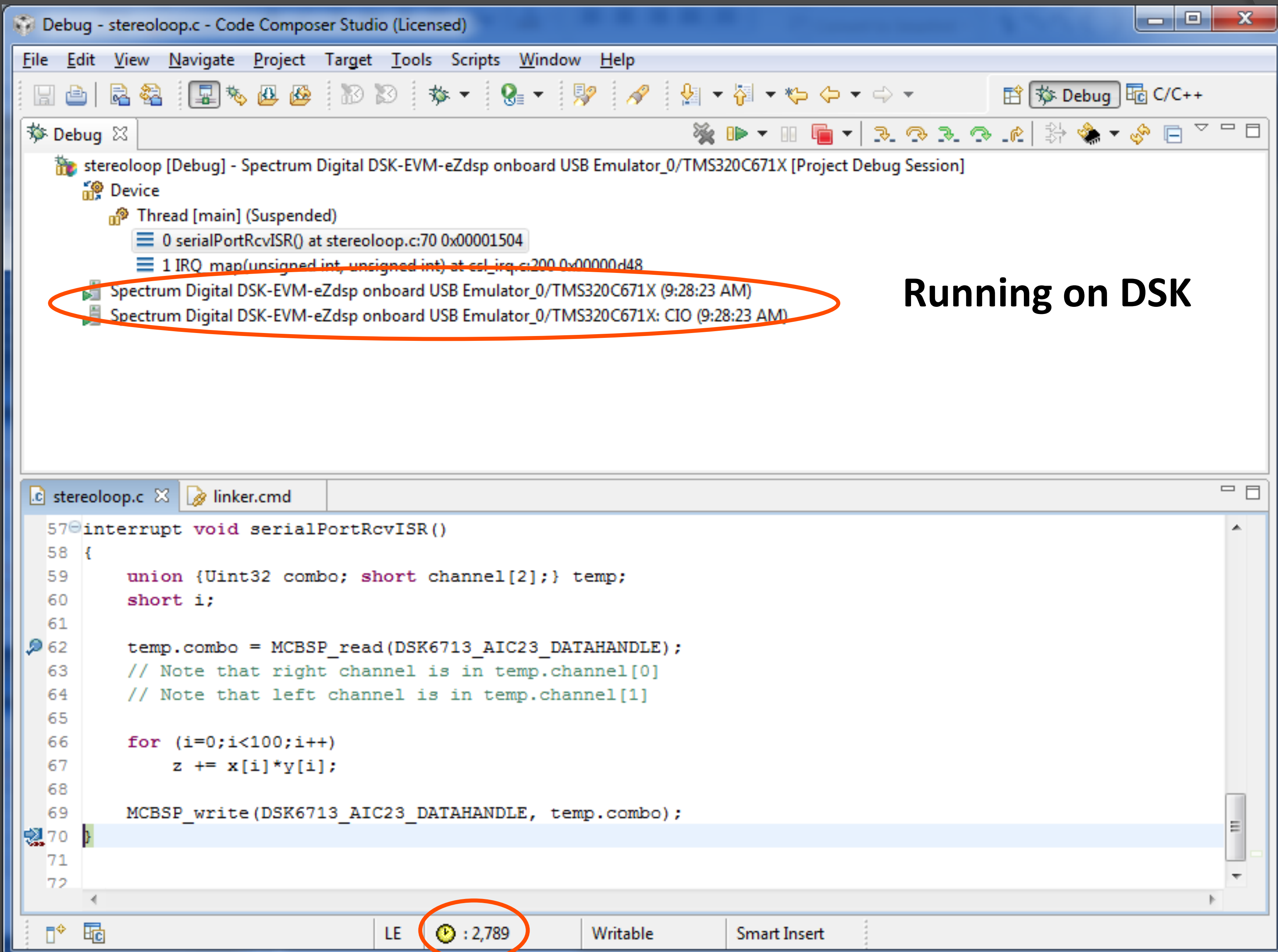
All calls to BSL functions removed

Code from ISR placed in a regular function called from main()

Tools -> Profile -> Setup, then Tools-> Profile -> View



Compare to Breakpoint/Clock Method

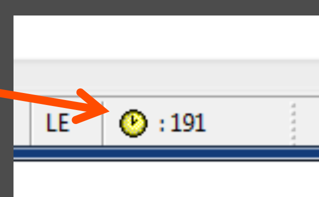


Simulator includes overhead of function call/return and allocating/deallocating local variables.

Other factors affecting code efficiency

- Memory
  - Code location (.text in linker command file)
    - internal SRAM memory (fast)
    - external SDRAM memory (typically 2-4x slower, depends on cache configuration)
  - Data location (.data in linker command file)
    - internal SRAM memory (fast)
    - external SDRAM memory (slower, depends on datatypes and cache configuration)
- Data types
  - Slowest execution is double-precision floating point
  - Fastest execution is fixed point, e.g. short

Example: Stereoloop project, changing .text and .data to external SDRAM:



About 2.5x slower than SRAM (can be worse)

TMS320C6000 C/C++Data Types

Type	Size	Representation	Range Minimum	Range Maximum
char, signed char	8 bits	ASCII	-128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	-32768	32767
unsigned short	16 bits	Binary	0	65535
int, signed int	32 bits	2s complement	-2147483648	214783647
unsigned int	32 bits	Binary	0	4294967295
long, signed long	40 bits	2s complement	-549755813888	549755813887
unsigned long	40 bits	Binary	0	1099511627775
enum	32 bits	2s complement	-2147483648	214783647
float	32 bits	IEEE 32-bit	1.175494e-38†	3.40282346e+38
double	64 bits	IEEE 64-bit	2.22507385e-308†	1.79769313e+308
long double	64 bits	IEEE 32-bit	2.22507385e-308†	1.79769313e+308

Final Remarks

- You should have enough information to complete Lab 1
  - Lab/lecture slides
  - Reference material noted in slides
  - Textbooks listed in syllabus
  - Please make sure you understand what you are doing. Please ask questions if you are unsure.
- Lab 1 Part 3: Signal Squaring
  - Simple example of (memoryless) non-linear signal processing
  - Sometimes used in synchronization algorithms
  - You want the analog input signal to use the full range of the ADC but avoid clipping (clipping = very bad nonlinear distortion)
  - You also want to avoid clipping in the output
  - Careful analysis of the output will reveal certain “features” of the AIC 23