

Contents

5. Memory Issues.....	5-1
5.1. Overview.....	5-1
5.2. Background information.....	5-1
5.2.1. Memory characteristics	5-1
5.2.2. About Scatter-loading.....	5-2
5.3. Instructions	5-3
5.3.1. Memory access pattern and scatter-loading.....	5-5
5.4. Exercise	5-9
5.5. Reference	5-10

5. Memory Issues

5.1. Overview

Memory configuration and usage are very important to embedded SoCs. This lab demonstrates the effects of different memory access patterns and data storage allocations. Moreover, often only a limited memory resource is present in a cost-sensitive embedded system, therefore mapping programs to limited memory resource is an essential skill. You will learn the followings:

1. Understand how to allocate data storages and arrange accesses.
2. Mapping memory using scatter-loading.

5.2. Background information

5.2.1. Memory characteristics

Memories in embedded systems are essentially divided into on-chip memories and off-chip memories according to their location in a system. On-chip memories are often implemented using SRAM, embedded DRAM, and ROM. Off-chip memories include flash, DRAM, and ROM. On-chip memories are often faster but also more expensive than off-chip memories. However, the high-density and less expensive feature of off-chip memories allow greater amount of data to be stored. Therefore on-chip memories are often incorporated to handle time-critical tasks, and are often of smaller size compare to off-chip memories; while off-chip memories are often adopted where data storage that needs great capacity and speed is not critical.

An important characteristic of DRAM is that sequential memory accesses requires less wait states than non-sequential accesses. This is because DRAM memory organization is addressed by row and column. The row address strobe (RAS) signal is first generated, then the column address strobe (CAS) is generated. If the next access is within the same row, no new row address is required to be provided. This enables faster data access and less power consumption.

Therefore ARM exploits the fact that most addresses (75%) are generated from address incrementer. A signal *seq* denoting an address is generated from the incrementer is sent to DRAM controller. Thus external logics can then check for row boundaries from previous access address. This captures most of the accesses which fall within the same DRAM row. Such accesses are referred to sequential accesses and have less wait states compare to non-sequential accesses.

In this lab, we will demonstrate how to store data efficiently by exploiting memory characteristics mentioned. Moreover, effects on performance due to different characteristic between SRAM and DRAM are also covered.

5.2.2. About scatter-loading

Scatter-loading allows the designer to map the codes and data in a program to certain memory space. The memory mapping is defined by writing a scatter-loading description file (.scf). The scatter-loading description file defines the regions of each object's RO, RW, and ZI. The linker reads the scatter-loading description file and maps the memory as defined in the file. Therefore we can map the parts that require faster memory accesses, such as mapping ISRs into SRAM.

A scatter-loading description file and its memory mapping are shown in Figure 1 and Figure 2 respectively. The first line specifies one load region LR_1 at base address 0x8000 with a maximum size of 256MB. Four execution regions: ALL, RWZI, HEAPS, and STACKS are defined. The first 3 execution regions (ALL, RWZI, and HEAPS) are mapped consecutively as defined by +0 in the field next to the execution region name. STACKS execution region has a stack base address at 0x1000000 and a maximum stack size of 0x0FFC0000. The statement inside the execution region defines which data should be mapped; for instance, `*(+RO)` in execution region ALL defines all RO code and data to be mapped into ALL region except those specified. Another example is to specify the mapping of certain data from a certain object, such as `heaps.o(+ZI)` maps the ZI data into HEAPS region. The term `UNINIT` tells the linker not to initialize the data, this can be used for uninitialized data or memory mapped I/O.

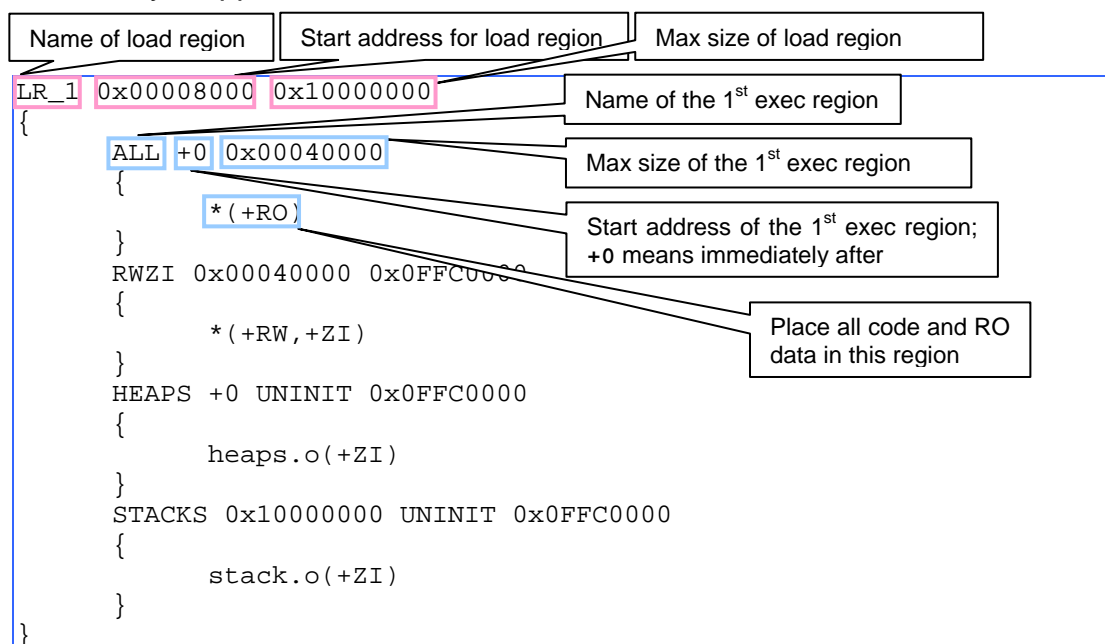


Figure 1 Scatter-loading description file – DRAM.scf

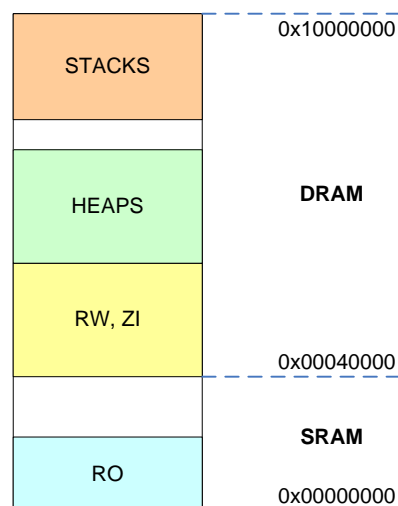


Figure 2 The memory map defined in DRAM.scf

The benefits of using a scatter description file are:

- All the (target-specific) absolute addresses chosen for your devices, code, and data are located in one file and maintenance is simplified.
- If you decide to change your memory map (for example if peripherals are moved), you do not have to rebuild your entire project but only to re-link the existing objects.

5.3. Instructions

The ARM program in this lab performs block reading from a larger data buffer (176x32) to a (4x4) block buffer, which is common in video applications. The program consists of 2 parts. The first part of the program stores the data in `data_array` in a linear fashion as shown in Figure 3, the address generation for the block access is complicated in this part. The other part stores the data in a block tile manner as shown in Figure 4, resulting in a simpler address generation.

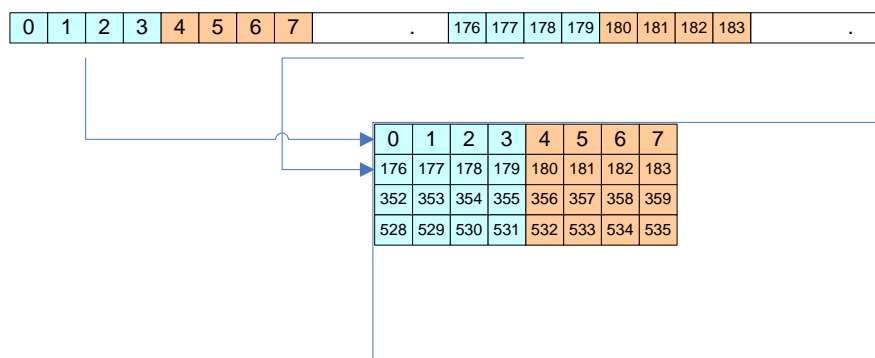


Figure 3 Block access for sequentially stored data

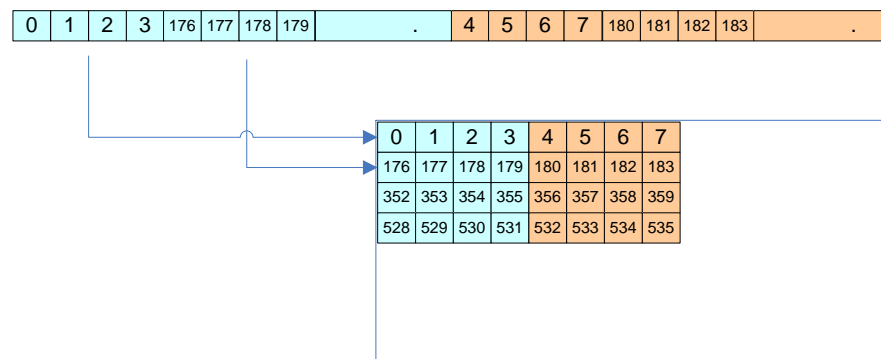


Figure 4 Block access for block tile stored data

```
for(iblk_Y = DATA_BLK_HEIGHT;iblk_Y != 0;iblk_Y--) {
    for(iblk_X = DATA_BLK_WIDTH;iblk_X != 0;iblk_X--) {
        for(row = 4; row != 0; row--) {
            memcpy(pblk,pdata_array_tmp,4*sizeof(unsigned short));
            pblk += 4;
            pdata_array_tmp += DATA_WIDTH;
        }
        //- update pointers to next block
        pdata_array += 4;
        pblk=blk_buff;
        pdata_array_tmp = pdata_array;
        //- computblock
        blk_average[blk_cnt] = compute_block_average(blk_buff);
        blk_cnt++;
    }
    pdata_array += DATA_WIDTH*3;    // next block row
}
```

Figure 5 Block access code for sequentially stored data

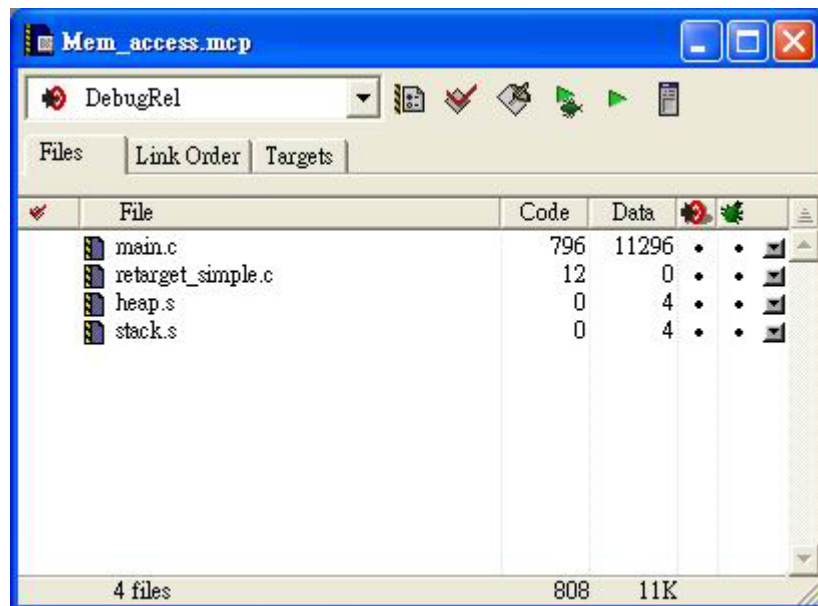
```
for(iblk_Y = DATA_BLK_HEIGHT;iblk_Y != 0;iblk_Y--) {
    for(iblk_X = DATA_BLK_WIDTH;iblk_X != 0;iblk_X--) {
        memcpy(pblk, pdata_array_tmp, 16*sizeof(unsigned short));
        pdata_array_tmp +=16;
        //- update pointers to next block
        pblk=blk_buff;
        //- computblock
        blk_average[blk_cnt] = compute_block_average(blk_buff);
        blk_cnt++;
    }
}
```

Figure 6 Block access code for block tile stored data

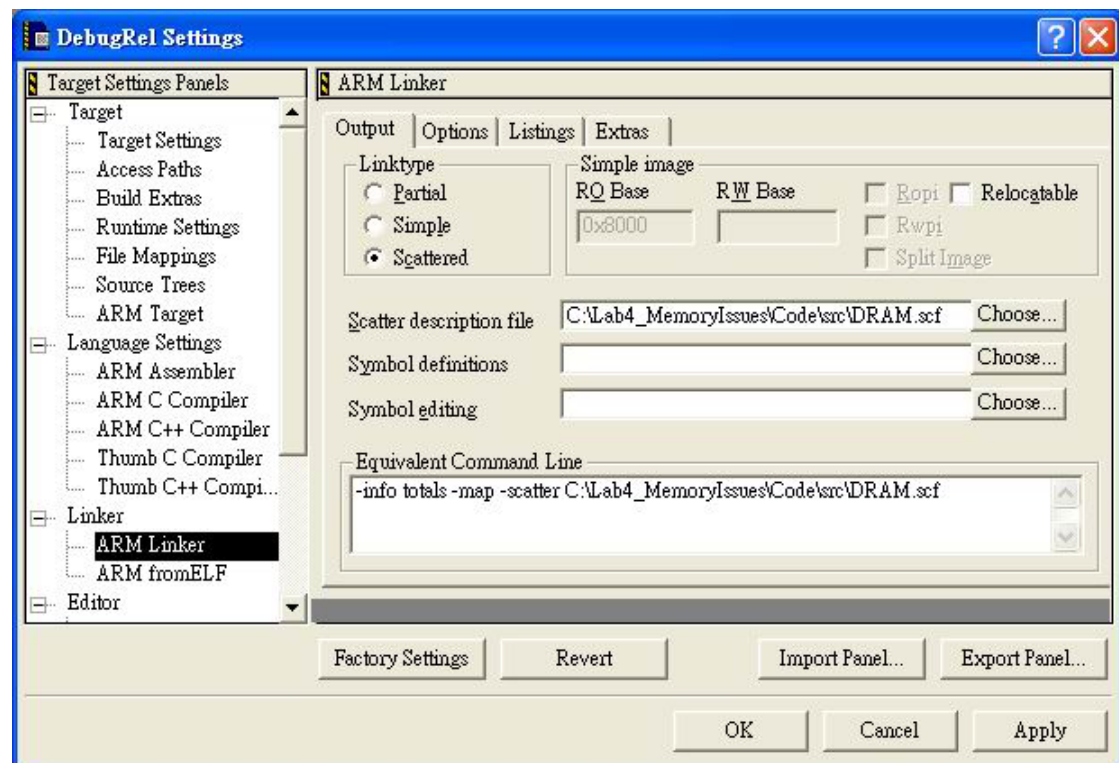
5.3.1. Memory access pattern and scatter-loading

1. Start CodeWarrior IDE
2. Select **File** → **Open** to open the project file **Mem_access.mcp**. This project has the following files:

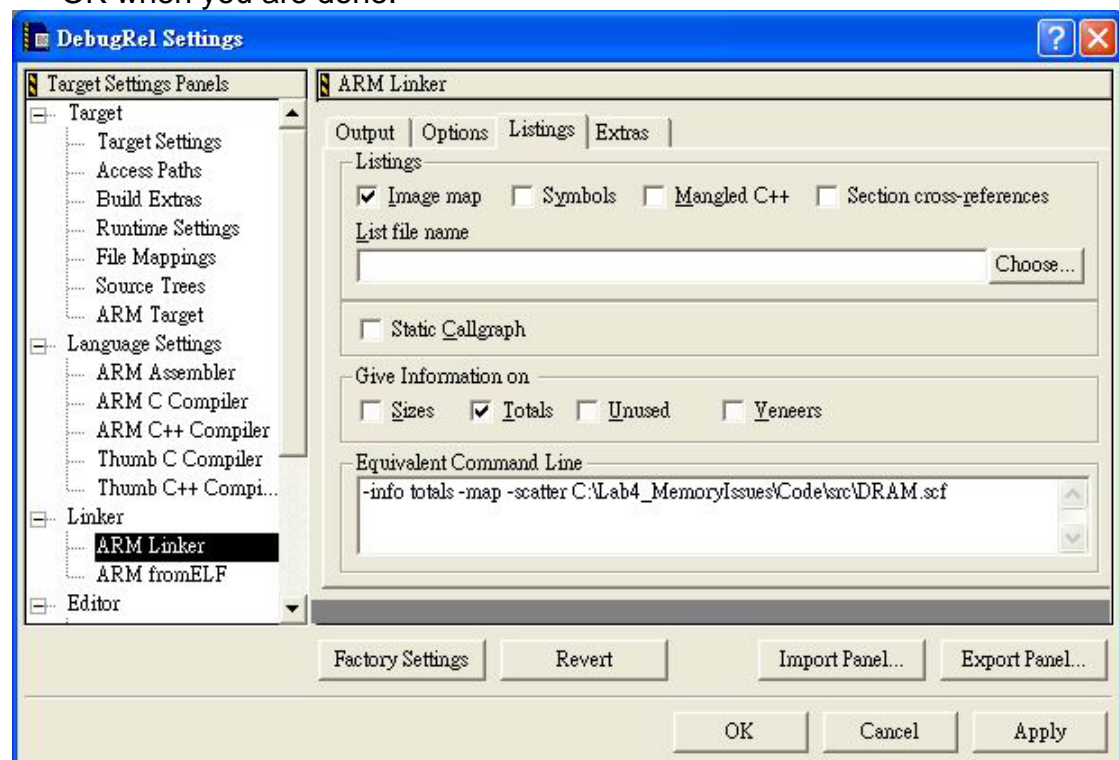
main.c	main program file
retarget_simple.c	retarget file for remapping heaps and stacks
heaps.s	assembly to export the bottom of heap
stacks.s	assembly to export the top of stack



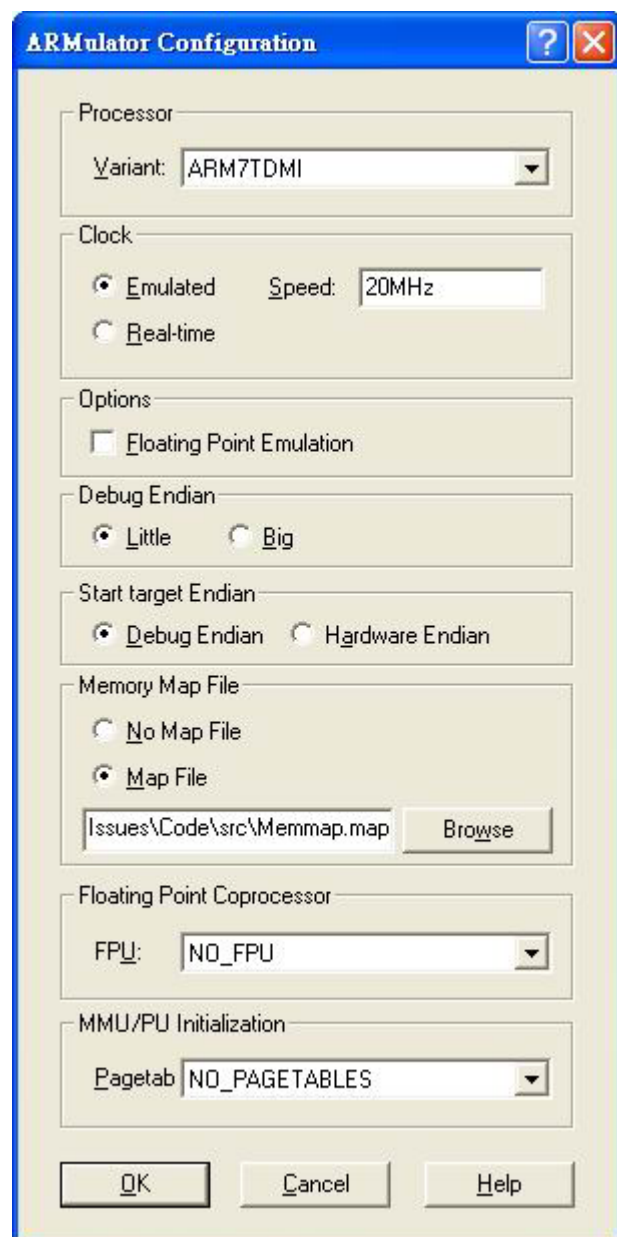
3. Press **DebugRelSettings** button on the **Project Management Window**, a Debug RelSettings window pops up.
4. Click on **ARM Linker** in **Target Settings Panel**. The current **Linktype** in ARM Linker panel is Simple.
5. Check **Scattered** check box in Linktype, specify Scatter-loading description file **DRAM.scf**. Press **Apply** button after you are done. This informs the linker to map the memory according to the descriptions in DRAM.scf. DRAM.scf maps all RW and ZI into DRAM region (0x00040000 ~ 0x10000000).



- Click on **Listings** tab in ARM Linker panel. Check the **Image map** check box to generate the image map after linking. You can specify the output file for the image map by inputting path and filename in **List file name**. Press OK when you are done.



7. Press **Make** button on the Project Management Window to build the project. The warning in the **Errors&Warning** is OK. The image map will be listed in the Errors&Warnings window. Please observe how the image map is organized.
8. Press **Run-debug** button on the Project Management Window to debug Mem_access.axf in AXD.
9. In AXD, **Options→ Config Target**. Select ARMUL in **Choose Target window** to configure ARMulator.
10. In **ARMulator Configuration window**, configure as follows then press OK.
Clock: check Emulated, Speed=20MHz
Memory map file: check Map File, **Memmap.map**



11. AXD will restart, when ask upon reloading the image, press **yes**. The **RDI Log** of the **System Output Monitor** shows the memory configuration.

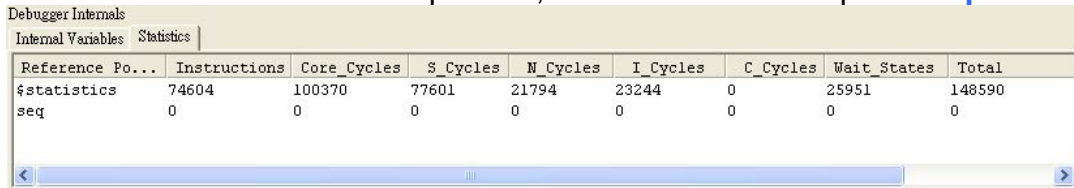


12. Press **F5** or **Go** button, the program halts at `int main(void)`. Set 4 breakpoints at line **58**, **76**, **100**, and **115**. The process in the first 2 breakpoints are block accesses for sequentially stored data, the process in the last 2 breakpoints are block accesses for block file stored data.

```

56     printf("Memory Access Example\n");
57
58     for(iblk_Y = DATA_BLK_HEIGHT; iblk_Y != 0; iblk_Y--) {
59         for(iblk_X = DATA_BLK_WIDTH; iblk_X != 0; iblk_X--) {
60             for(row = 4; row != 0; row--) { //copy from data_array into blk
61                 memcpy(pblk, pdata_array_tmp, 4*sizeof(unsigned short));
62                 pblk += 4;
63                 pdata_array_tmp += DATA_WIDTH; // block access pattern dt
64             }
65             //- update pointers to next block
66             pdata_array += 4;
67             pblk=blk_buff;
68             pdata_array_tmp = pdata_array;
69             //- computblock
70             blk_average[blk_cnt] = compute_block_average(blk_buff);
71             blk_cnt++;
72         }
73         pdata_array += DATA_WIDTH*3; // next block row
74     }
75
76     printf("blk_average[0]= %d\n\n", blk_average[0]);
77
78
98     blk_cnt=0;
99
100    for(iblk_Y = DATA_BLK_HEIGHT; iblk_Y != 0; iblk_Y--) {
101        for(iblk_X = DATA_BLK_WIDTH; iblk_X != 0; iblk_X--) {
102            memcpy(pblk, pdata_array_tmp, 16*sizeof(unsigned short));
103            pdata_array_tmp += 16; // block access pattern due to block s
104            //- update pointers to next block
105            //pdata_array += 16; //no need!!
106            pblk=blk_buff;
107            //pdata_array_tmp = pdata_array; //no need!!
108            //- computblock
109            blk_average[blk_cnt] = compute_block_average(blk_buff);
110            blk_cnt++;
111        }
112        //pdata_array += DATA_WIDTH*3; // next block row no_need!!
113    }
114
115    printf("blk_average[0]= %d\n\n", blk_average[0]);
116
```

13. Press **F5** or **Go** again to run to line **58** (1st breakpoint). Open Debugger Internals by pressing **ALT-D** or from menu **System Views → Debugger Internals**.
14. Click on **Statistics** tab in Debugger Internals panel, then **right mouse click** in Debugger Internals panel to access the drop menu. Click **Add New Reference Point** in drop menu, name the reference point **seq**.



Reference Po...	Instructions	Core_Cycles	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Wait_States	Total
\$statistics	74604	100370	77601	21794	23244	0	25951	148590
seq	0	0	0	0	0	0	0	0

15. Press **F5** or **Go** button, the program will halt at line **76** (2nd breakpoint). Record the statistics in Debugger Internals for reference point **seq**. These statistics represents the statistics for block accessing sequentially stored data.
16. Press **F5** or **Go** again to run to line **100** (3rd breakpoint). Add a new reference point named **blk** in statistics of Debugger Internals.
17. Press **F5** or **Go** button, the program will halt at line **115** (4th breakpoint). The statistics in reference point **blk** represents the process of block accessing block tile stored data. Record the **blk** statistics and **compare** with the previously recorded **seq** statistics.
18. The **NS cycles** in **blk** case should be less than the NS cycles in **seq** case. The **wait states** for **blk** case should also be less than that of **seq** case. Please try to explain this result.

5.4. Exercise

First, try to show the address pointed by `pblk` and `pdata_array`. Identify which regions are the two arrays (`blk_buff`, `data_array`) located in. (**Hint**: move the cursor to the pointers or use add watch)

Then try to load all RW and ZI of `main.o` into SRAM region (0x8000~0x40000). Rerun the simulation and compare the statistics results for **seq** and **blk** case. Compare the address pointed by pointer again with previous case.

Finally, try move the declaration of `blk_buff` and `data_array` into `main()` function. Compare the address pointed by pointer again with previous cases. (**Hint**: observe the value of `r13`)

5.5. Reference

1. Using Scatter-loading for..., ADS Developers Guide [DUI0056D, 6.6 6.7 6.8 6.9]
2. Using Scatter-loading Description Files, ADS Linker and Utilities Guide [DUI0151A, 5]
3. Specify code from C and C++, ADS Compilers and Libraries Guide [DUI0067D]
4. Memory Map, ADS Debug Target Guide [DUI0058D, 2.8]
5. Steve Furber, "ARM System-on-Chip Architecture," Addison Wesley, 6.9, 8.1, 2000.