Application Note

Stack Overflow Detection on UCD3138 Devices

Xuemei Lu

ABSTRACT

It is hard to tell how much size a stack uses statically since the size varies with the code running. There are other things that are stored in RAM as well, for example global variables. If a stack overflow happens, it modifies the others unexpectedly, causing unpredictable problems. Therefore, it is necessary to reserve enough room for STACK. This application note describes two processes to check if a stack overflow happens.

Table of Contents

1 Introduction........2
1.1 Check the Size for Each Stack........2
2 Check if an Overflow Happens........3
3 Summary........5
4 References........5

List of Figures

Figure 1-1. Stack Allocation........2
Figure 2-1. Memory Peek/Poke........3
Figure 2-2. Stack Usage Detection With Memory Peek/Poke........4

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

1.1 Check the Size for Each Stack

There are generally four stacks used in demo codes: user stack, IRQ stack, FIQ stack and supervisor stack. User stack is for background routines, IRQ stack is for standard interrupt routines, FIQ stack is for fast interrupt routines, and supervisor stack is for software interrupt (SWI) routines. Some can have undefined stack and abort stack for exceptions, but they are rarely used in normal cases. The stacks are declared at the top of the load.asm file. Take the following as an example to see how each stack is allocated.

```
SUP_STACK_TOP  .equ    0x6bffc ;Supervisor mode (SWI stack) starts at top of memory
FIQ_STACK_TOP    .equ    0x6be00 ;allocate 256 bytes to supervisor stack, then do FIQ stack
IRQ_STACK_TOP    .equ    0x6bd00 ;allocate 256 bytes to fiq stack, then start irq stack
USER_STACK_TOP    .equ    0x6bb00 ;Allocate 512 bytes to irq stack, regular stack gets rest, down
to variables
        .global _c_int00
        .global $c_int00
```

With those definitions, this shows that the top of user stack is at address 0x6bb00 and down to variables, IRQ stack is allocated from address 0x6bb00 (bottom) to 0x6bd00 (top), FIQ stack is allocated from address 0x6bd00 (bottom) to 0x6be00 (top), and supervisor stack is allocated from address 0x6be00 (bottom) to 0x6bffc (top).

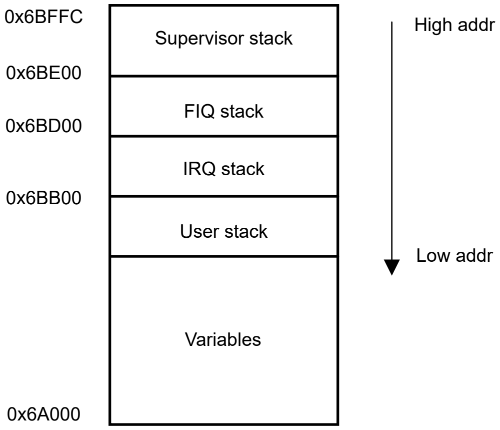


Figure 1-1. Stack Allocation

As for the bottom of user stack, check the .map file, which shows the RAM memory allocation. The .map file is generated by CCS when building the firmware project, and shall be located in the same directory as .x0 file. Following is an example copied from a .map file, and the text shows the variables starts from address 0x6a000 and ends at 0x6a80d. Therefore, the user stack can be down to 0x6a80e.

```
        name                origin    length      used     unused   attr    fill
----------------------   --------  ---------   --------  --------   ----  --------
  FLASHVECS              00000000   00000020  00000020  00000000   R  X
  PFLASH                 00000020   00007f34  00003d1e  00004216   R  X
  DEVICEID               00007f54   00000020  0000001f  00000001   R  X
  FIXTFA                 00007f74   00000004  00000000  00000004   R  X
  FIXCONST               00007f78   00000080  00000000  00000080   R  X
  FLASHSUM               00007ff8   00000008  00000000  00000008   R  X
  ROMVECS                00020000   00000020  00000000  00000020   RWIX
  ROM                    00020020   00001d5e  00000000  00001d5e   RWIX
  SINE                   00021d7e   00000282  00000000  00000282   RWIX
  DFLASH                 00069800   00000800  00000398  00000468   R  X
  RAM                    0006a000   00001dd0  0000080d  000015c3   RW
  RAM_PGM_AREA           0006bdd0   00000080  00000000  00000080   RW
  STACKS                 0006be50   000001b0  00000000  000001b0   RW
  LOOP_MUX               00120000   00000070  0000006c  00000004   RWIX
```

In some demo codes, there is stack allocation in the .cmd file, in the following. The stack allocation does not take effect, stack allocation is actually done in load.asm.

```
STACKS    (RW) : org = 0x0006BE50, len = 0x000001B0
.stack         : {                              /* total = 400 = 0x190               */
                        _StackUSER_  = .            + 184;   /* USER                      */
                        _StackFIQ_   = _StackUSER_  + 112;   /* FIQ                       */
                        _StackIRQ_   = _StackFIQ_   + 84;    /* IRQ                       */
                        _StackABORT_ = _StackIRQ_   + 4;     /* ABORT                     */
                        _StackUND_   = _StackABORT_ + 4;     /* UND                       */
                        _StackSUPER_ = _StackUND_   + 12;    /* SUPER                     */
                     } > STACKS                          /* Software System stack     */.
```

2 Check if an Overflow Happens

Stacks are part of RAM, and RAM is entirely initialized to be zeros in the load.asm. To check the usage for each stack, one option is to read from the stack location and see if there is all-zero space from the bottom of a stack. It is necessary to check the stacks while the code is running, and this can be done with an adapter and the Memory Peek/Poke tool that is embedded in the UCD3xxx Device GUI.

Connect the adapter, launch UCD3xxx Device GUI, go to Debug > Memory Peek/Poke.

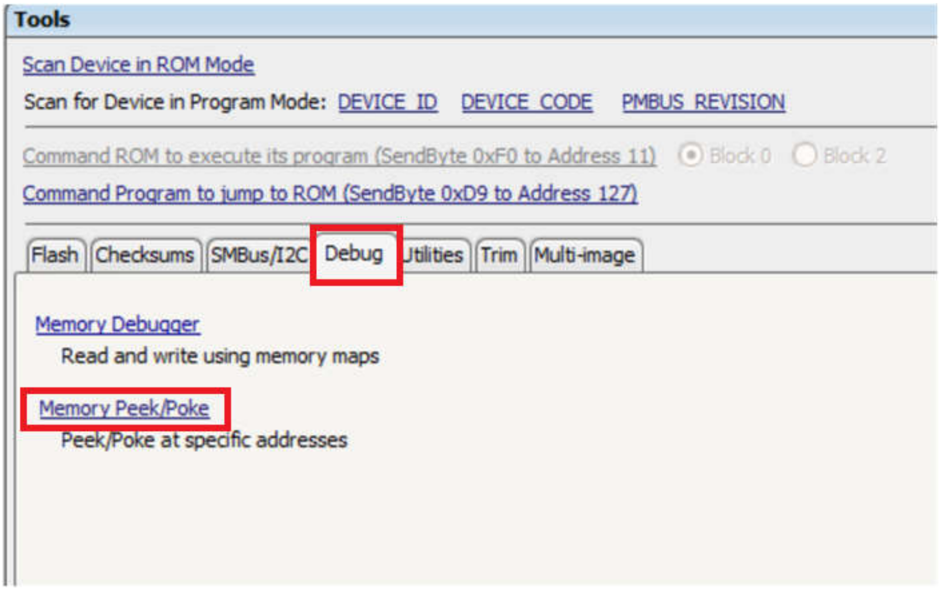
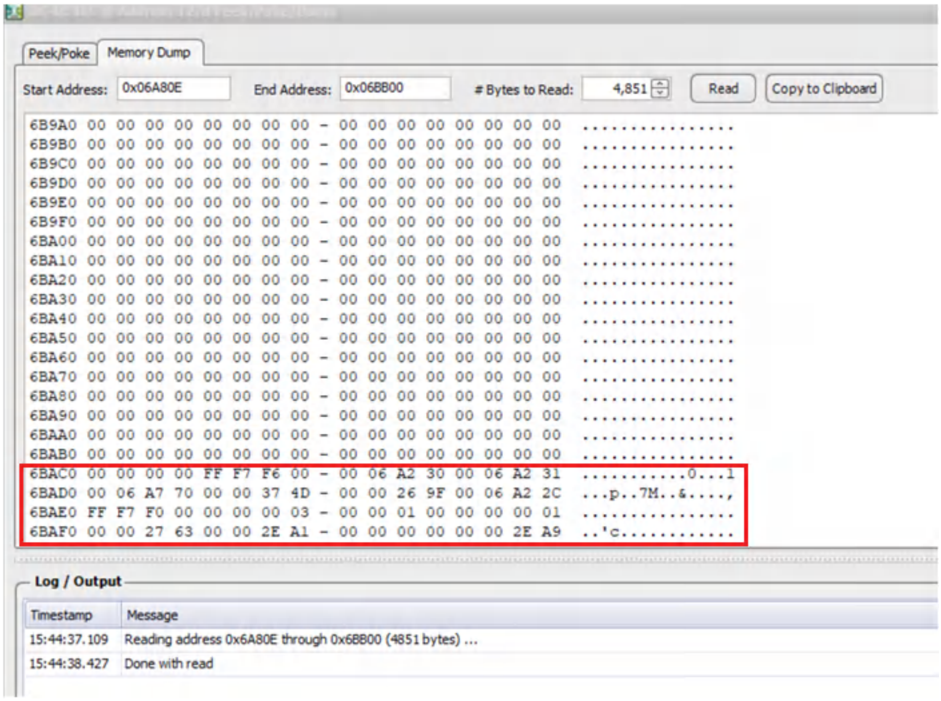

Tools
Scan Device in ROM Mode
Scan for Device in Program Mode: DEVICE_ID DEVICE_CODE PMBUS_REVISION
Command ROM to execute its program (SendByte 0xF0 to Address 11)
Block 0
Block 2
Command Program to jump to ROM (SendByte 0xD9 to Address 127)
Flash
Checksums
SMBus/I2C
Debug
Utilities
Trim
Multi-image
Memory Debugger
Read and write using memory maps
Memory Peek/Poke
Peek/Poke at specific addresses

Figure 2-1. Memory Peek/Poke

Take user stack as an example. Read the memory from address 0x6a80e to 0x6bb00 like the following. It shows about 60 bytes are used and 4754 bytes are still available. Of course, it is hard to capture the worst case this way, since the stack varies from time to time during the code running. But it should give us some clues whether an overflow could possibly happen by checking how much margin it has.



Start Address: 0x06A80E   End Address: 0x06BB00   # Bytes to Read: 4,851

```
6B9A0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6B9B0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6B9C0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6B9D0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6B9E0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6B9F0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA10 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA20 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA30 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA40 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA50 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA60 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA70 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA80 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BA90 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BAA0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BAB0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
6BAC0 00 00 00 00 FF F7 F6 00 - 00 06 A2 30 00 06 A2 31  ...........0...1
6BAD0 00 06 A7 70 00 00 37 4D - 00 00 26 9F 00 06 A2 2C  ...p..7M..&....,
6BAE0 FF F7 F0 00 00 00 00 03 - 00 00 01 00 00 00 00 01  ................
6BAF0 00 00 27 63 00 00 2E A1 - 00 00 00 00 00 00 2E A9  ..'c............
```

Log / Output

```
Timestamp     Message
15:44:37.109  Reading address 0x6A80E through 0x6BB00 (4851 bytes) ...
15:44:38.427  Done with read
```

Figure 2-2. Stack Usage Detection With Memory Peek/Poke

Another option is to add test code in the firmware to continuously check if the bottom (or near the bottom for some margins) of the stack is non-zero. Once a non-zero detected, it shows an overflow happens, toggle an IO for report. The benefit for this option is that it checks continuously during the code running.

Take the supervisor stack as an example, the detection code can be similar to the following, in which the stack_mon.ptr is a pointer pointing at the bottom of supervisor stack initially. It is required to start from the bottom.

```
                if (((Uint32)stack_mon.ptr == (Uint32)SUP_STACK_TOP) || (Uint32)(*(stack_mon.ptr) !
= 0))
                {
                    // reached top of stack so the stack is all zeros (so stack is empty), or else
encountered the stack (as encountered a non-zero word)
                    stack_sup_headroom = (int32)stack_mon.ptr - (int32)SUP_STACK_BOT;
                    if (stack_sup_headroom < STACK_MON_HEADROOM_ALERT)
                    {
                        LoopMuxRegs.DTCIOCTRL.bit.DTC_B_GPIO_VAL = 1;
                    }
                }
                else
                {
                    // move onto the next address in the stack
                    stack_mon.ptr++;
                }
```

3 Summary

Unexpected problems can happen when stack overflow occurs. This application note describes two options for stack overflow detection. Option 1 is done with memory peek/poke. It is easy to implement, but cannot detect the worst case because stack usage varies with code running. Option 2 is detecting continuously and alerts once an overflow happens.

4 References

- Texas Instruments, *Fusion Digital Power Studio*.

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