VINCULUM

BINDING USB TECHNOLOGIES

Future Technology Devices International Ltd.

Vinculum II Memory Management Application Note AN_157

Document Reference No.: FT_000352

Version 1.2

Issue Date: 2010-11-26

This application note discusses methods for efficient use of RAM in Vinculum-II (VNC2) applications.

Future Technology Devices International Ltd (FTDI)

Unit 1, 2 Seaward Place, Centurion Business Park, Glasgow, G41 1HH, United Kingdom

Tel.: +44 (0) 141 429 2777   Fax: + 44 (0) 141 429 2758

E-Mail (Support): support1@ftdichip.com

Web: http://www.ftdichip.com

Use of FTDI devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold harmless FTDI from any and all damages, claims, suits or expense resulting from such use.

Copyright © 2010 Future Technology Devices International Limited

1 Introduction

The FTDI Vinculum-II (part number VNC2) is a dual channel USB 2.0 Device/Host, microcontroller based IC targeted at USB applications. The device has 256kB of ROM storage and 16kB of user RAM.

Memory for storing global and static variables in RAM is allocated by the Vinculum-II Toolchain which is used to develop user application firmware. This Toolchain is available free of charge at the following website **http://www.ftdichip.com/Firmware/VNC2tools.htm#VNC2Toolchain**.

The Vinculum Operating System (VOS) kernel allows multiple threads to have a separate and configurable stack space and also maintains a global heap for dynamic memory allocation within VNC2.

This application note discusses how to efficiently allocate the VNC2 memory between threads and dynamic allocation.

This application note discusses types of memory allocation, the sizing of threads and then explains possible error scenarios.

Table of Contents

2 Understanding the MAP file

Before discussing Memory allocation, it is important to understand the MAP file and how this influences memory allocation.

A map file is generated by the VNC2 linker (VinL.exe) and is used to help analyse the RAM and ROM memory usage.

2.1 Segment Map

A segment map is included in the MAP file which has 5 sections.

```
=================================================================================
SEGMENT_MAP
=================================================================================
Start Address                   Segment Name                    Segment Size
=================================================================================
0003c0                          .text                           000001edae
01f16e                          .dataFlash                      000000023a
000000                          .dataRAM                        000000023a
00023a                          .bss                            0000000adc
000d16                          .HeapRAM                        0000002ee8
```

Figure 1 MAP file Segment Map

The .text and .dataFlash segments are sections which are stored in ROM.

- .text section is the program code storing the application
- .dataFlash is the initialised variables block, this is copied to RAM when the application starts

The .dataRAM, .bss and .HeapRAM sections refer to RAM storage.

- .dataRAM – initialised variable storage for globals, static variables and certain initialised local variables
- .bss – uninitialized variable storage
- .HeapRAM – available memory for the heap (dynamically allocated storage)

Sections here are sized by the linker at compile time.

2.1.1 Initial Stack Size

The total size of the three RAM storage areas does not equal the size of the RAM because a certain amount of RAM is required, and reserved, for the stack for the main() function. This is, by default, 1024 bytes but can be changed with the –k command line switch when calling the linker or altering the "Stack Size" option for the linker in VinIDE.

The main() function will require stack space to perform initialisation but this memory is not used after the VOS kernel starts (when vos_start_scheduler() is called). The size of the initial stack size should be large enough to perform initialisation but not too large to use memory that is required by the application threads.

In VOS kernel versions up-to and including V1.2.2, this memory is not recovered however in V1.2.4 and later this memory can be reused by the application heap.

NOTE: It is not recommended to make local variables in the main() function which persist when the kernel starts on the vos_start_scheduler() call.

2.2 Symbol Table

The map file will list all global and static variables which are created in the application but not those allocated from any archive files which are linked to the application.

```
=================================================================================
SYMBOL_TABLE
=================================================================================
SYM ADDRESS PUBLIC SYM NAME       SYM TYPE    MEM TYPE FILE NAME         SYM SIZE
NOTE:L=LABEL F=FUNCTION 0=FLASH and 1=RAM
=================================================================================
00043a main                           F              0 tm123.obj            007ce
000c08 firmware                       F              0 tm456.obj            00dde
00023c tcbMonitor                     L              1 tm456.obj            00002
00023e monInterface                   L              1 tm456.obj            00001
00023f hUsb1                          L              1 tm456.obj            00002
```

Figure 2 MAP file Symbol Table

The symbols with a “MEM TYPE” of 0 are in ROM and 1 are in RAM. In the example above, global variables are tcbMonitor, monInterface and hUsb1.

3 Types of Memory Allocation

3.1 Fixed Memory

Globals and static variables, as well as some initialised local variables, are allocated a fixed block at the start of RAM by the linker. The size of this RAM is obtained from the combined size of the .dataRAM and .bss segments.

In this section initialised variables are allocated first, followed by uninitialized variables.

The kernel and some drivers use global variables to store persistent values. This storage is contained within the .dataRAM and .bss sections of the SEGMENT MAP in the map file.

3.2 Thread Memory

Each thread created by VOS is allocated its own stack in RAM when vos_create_thread() is called. The stack size for the thread is the size of the TCB (Thread Control Block) passed to the create thread function minus a small amount (46 bytes) for thread control information.

A thread's TCB is allocated from the .HeapRAM section dynamically at runtime. The thread must not overflow its stack.

Certain drivers will create threads to perform tasks. These threads will have a definition in their header file describing how much stack space the threads will require.

3.3 Dynamically Allocated Memory

An application is able to call malloc() to allocate memory from the remainder of the heap. If there is insufficient heap space left then the call will return NULL. A call to free() will release the memory back on to the heap to be re-used by subsequent malloc() calls.

3.4 Effect of Optimisation Levels in the Compiler

The compiler has several optimisation levels. Normally for a debug build optimisations will be turned off (-o 0) and for release builds this will be turned on (-o 4). The command line options are explained in the Vinculum-II User Guide document.

With optimisation off, the stack usage of code may be significantly larger than when optimisation is turned on. This is because in higher optimisation levels, unused code and variables are removed and memory used for some variables and intermediate calculations is reused. This makes the memory usage profile different for release and debug builds.

3.4.1 Reducing Stack Space while Debugging

A work around would be to use optimisation level 1. This includes most of the memory usage optimisations enabled for release builds but still allows (with several conditions) stepping through code.

In this level, however, it is not always possible to view variables with the debugger watch window and the values reported there may not be updated when expected. Expressions related to any variable that is removed through the optimisation may also be removed from code and therefore cannot have a breakpoint placed on them.

A possible method of improving code readability is to have a conditional definition of stack size for a thread in release and debug modes. This could be used to reduce the memory available to a thread in release mode while still allowing debugging to occur.

4 Sizing Threads

The amount of stack space allocated to a thread is determined heuristically. Most threads will use between 256 bytes and 4kB for their stacks. The space required is dependent on the local variable usage of the thread, This includes local variables of all functions called within the scope of the thread.

In a typical application there will be a mix of all types of memory usage described in previous chapters. The proportions will vary depending on the tasks performed and the style of the programmer.

While running, some threads will use and release stack space rapidly when calling other functions but use only a small amount for most of the time when they are running. Therefore it's important to find the maximum usage of a thread and size the memory allocated to it accordingly.

4.1 Procedure

The accepted procedure for arriving at the optimum stack size for a thread is to allocate an arbitrary amount of memory, usually 512bytes or 1kB initially.

If the application runs successfully then reduce the amount of memory by 64 or 128 bytes. This can be repeated until the application fails.

If, however, the application fails then increase the amount of stack space by 256 or 512 bytes until the application works. Once this has been achieved then the stack space can be reduced by small amounts again until the optimum size is found.

An application which fails due to lack of stack space for a thread will appear to halt and become unresponsive. All threads which are running, and the kernel, will stop when this occurs.

The kernel's interrupt handling and task switching functions use stack space reserved for the kernel and need not be factored in.

4.2 Future Improvements

At the time of writing this application note, FTDI are planning to introduce a Plugin for VinIDE called *Thread Manager* with the next release of their VNC2 Toolchain. This Plugin will show all threads which are used in the kernel and their memory usage. The memory usage will be represented as a current usage and a 'high-water mark' usage. Using this information it will be easier to efficiently calculate memory allocation for each thread.

5 Failure Modes

Quite often a buffer overrun will result in a kernel “oops”. This is a special feature of the VOS kernel where it detects if the memory for a thread or the kernel’s internal state has been corrupted.

There is no fixed, predictable failure mode for applications. Behaviour when buffers are overrun is typically unpredictable.

5.1 Fixed Memory

If a global or static memory buffer is overrun then the typical failure mode is that other global or static variables are affected. Another side effect may be a kernel “oops” since the kernel state variables are stored in this segment. Serious overruns will affect the thread memory and hence the behaviour of the kernel.

The linker will detect whether a global and static variable allocation exceeds the available RAM on the device.

5.2 Thread Memory

A thread stack overrun will cause a kernel “oops”. This will result in the application becoming unresponsive and all threads will cease to run.

5.3 Dynamically Allocated Memory

If a dynamically allocated buffer is overrun then neighbouring variables will be affected and subsequent calls to malloc() or free() may result in a kernel oops or unpredictable behaviour.

5.4 main() Function Stack

If the stack space allocated to the main() function is exceeded while actually in the main() function, or calling any other function from main() then an exception will occur.

The stack for main() is set to 0x400 bytes by default in the linker. If a large amount of dynamic memory is required by drivers and the main function exceeds the limits of it’s stack then there will be issues. This is normally noticed by a call to a driver init function (fat_init(), usbhost_init()) failing to return to the main() function.

To overcome this, either increase the stack space allocated to the main() function (-k options in the linker) or decrease the memory allocated to threads and buffers.

A point to note is that in V1.2.2 and older versions of the toolchain, the stack space for the main() function is not reclaimed when the kernel scheduler starts.

Copyright © 2010 Future Technology Devices International Limited

6 Hints and Tips

The following hints and tips have been collated concerning memory usage:

6.1 FAT File System Driver

The FAT file system requires about 1.5kB of RAM for persistent storage for operation. This is added to the .bss segment and not explicitly listed in the map file.

6.2 USBHost Driver

6.2.1 USBHost Threads

Three threads are required by this driver. The total amount of mempory allocated to these threads is listed in the "`USBHOST_MEMORY_REQUIRED`" definition in the USBHost.h header file.

6.2.2 USBHost Structures

The usbhost_init() function will reserve memory for USB interfaces, endpoints and transfer data depending on the parameters passed to the function. A formula for calculating how much memory is required is shown in the help file page for usbhost_init().

If a large number of interfaces are specified then this can use a significant amount of memory. For efficient memory usage please size the number of interfaces and endpoint structures to suit the deviceis used in your application.

6.3 Defining Arrays and Strings

The VinC compiler will store string literals in the global RAM and they can be referenced from anywhere in the program. For instance, a pointer to a string literal passed from a function is still valid after the program returns to the calling function. This pointer will also be valid when passed to another unrelated function. It is good practice to store pointers to string literals with the const keyword to allow the compiler to detect attempts to modify the string literal. This applies to both global and locally defined string literals.

The example in Figure 3 demonstrates string literal usage when returning a value from a function.

```
const char *st0 = "global char pointer"; // Global string literal

const char *strptr(void);

void main(void)

{

        const char *st1;

        st1 = strptr();

        st1[0] = '\0'; // (error) C2101 cannot modify a const object

}

const char *strptr(void)

{

        return "local char pointer"; // Local string literal

}
```

Figure 3 Example code for returning string literal from function

The situation is different for arrays. Although arrays may store strings they behave just like any other variable. A local array is initialised when a function is called and its scope will follow that of any other variable, even if it stores a 'string'. It is not possible to pass a pointer to a locally defined array back to a calling function as the storage for the array will be relinquished when the program returns from a function.

The initialised array will be recreated from data stored in ROM each time the array comes into scope. The address of the array cannot therefore be guaranteed to be at the same address each time, nor can it be expected that the data stored in the array is correct when the array goes out of scope. A function erroneously returning an array is shown in Figure 4.

```
char *strarr(void)
{
        char ar1[] = "local array"; // Local array
        return ar1; // not allowed - storage for ar1 is destroyed on return from function
}
```

Figure 4 Returning a pointer to an array from a function

Understanding this difference in behaviour can be exploited to save RAM memory by defining certain strings as arrays and hence storing them in ROM.

If you are using string literals locally in functions then define them as arrays "`char x[] = "string";`" This is the most efficient (for RAM) way of storing strings. If the string literal is to be passed between functions or even assigned to a global pointer in a function then it must be defined as a string literal "`char *x = "string";`".

6.4 Stack Reclaiming

In V1.2.2 and older versions of the toolchain, the kernel idle stack and the main() function were allocated fixed amounts of memory for their operation. Versions V1.2.4 onwards will automatically reclaim the memory allocated to the main() function for stack space and allow the size of the stack for the idle thread to be altered.

The stack allocated to main() is specified with the –k command line parameter in the linker.

7 Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place,
Centurion Business Park
Glasgow, G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

E-mail (Sales) sales1@ftdichip.com
E-mail (Support) support1@ftdichip.com
E-mail (General Enquiries) admin1@ftdichip.com
Web Site URL http://www.ftdichip.com
Web Shop URL http://www.ftdichip.com

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited (Taiwan)
2F, No 516, Sec. 1 NeiHu Road
Taipei 114
Taiwan, R.O.C.
Tel: +886 (0) 2 8791 3570
Fax: +886 (0) 2 8791 3576

E-mail (Sales) tw.sales1@ftdichip.com
E-mail (Support) tw.support1@ftdichip.com
E-mail (General Enquiries) tw.admin1@ftdichip.com
Web Site URL http://www.ftdichip.com

Branch Office – Hillsboro, Oregon, USA

Future Technology Devices International Limited (USA)
7235 NW Evergreen Parkway, Suite 600
Hillsboro, OR 97123-5803
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

E-Mail (Sales) us.sales@ftdichip.com
E-Mail (Support) us.support@ftdichip.com
E-Mail (General Enquiries) us.admin@ftdichip.com
Web Site URL http://www.ftdichip.com

Branch Office – Shanghai, China

Future Technology Devices International Limited (China)
Room 408, 317 Xianxia Road,
ChangNing District,
ShangHai, China

Tel: +86 (21) 62351596
Fax: +86(21) 62351595

E-Mail (Sales): cn.sales@ftdichip.com
E-Mail (Support): cn.support@ftdichip.com
E-Mail (General Enquiries): cn.admin1@ftdichip.com
Web Site URL http://www.ftdichip.com

Distributor and Sales Representatives

Please visit the Sales Network page of the FTDI Web site for the contact details of our distributor(s) and sales representative(s) in your country.

Appendix A – References

Vinculum II datasheet

http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_Vinculum-II.pdf

Application and Technical Notes available at http://www.ftdichip.com/Support/Documents/AppNotes.htm

Vinculum-II User Guide

http://www.ftdichip.com/Support/Documents/AppNotes/AN_151_Vinculum-II_User_Guide.pdf

Vinculum-II Toolchain Installation Guide

http://www.ftdichip.com/Support/Documents/AppNotes/AN_145_Vinculum-II_Toolchain_Installation_Guide.pdf

Vinculum-II Toolchain Getting Started Guide

http://www.ftdichip.com/Support/Documents/AppNotes/AN_142_Vinculum-II_Tool_Chain_Getting_Started_Guide.pdf

Vinculum-II Debug Interface Description

http://www.ftdichip.com/Support/Documents/AppNotes/AN_138_Vinculum-II_Debug_Interface_Description.pdf

Vinculum-II Errata Technical Note

http://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_118_VNC2%20Errata%20Technical%20Note.pdf

Vinculum II Toolchain (IDE)

http://www.ftdichip.com/Firmware/V2TC/VNC2toolchain.htm

FT_Prog

http://www.ftdichip.com/Support/Utilities/FT_Prog_v1.10.zip

Appendix B – Revision History

Version 1.0	First release	28th October 2010
Version 1.1	Added section 3.4	1st November 2010
Version 1.2	Added section 5.4 and section 6	26th November 2010

Appendix C Legal Disclaimer:

System and equipment manufacturers and designers are responsible to ensure that their systems, and any Future Technology Devices International Ltd (FTDI) devices incorporated in their systems, meet all applicable safety, regulatory and system-level performance requirements. All application-related information in this document (including application descriptions, suggested FTDI devices and other materials) is provided for reference only. While FTDI has taken care to assure it is accurate, this information is subject to customer confirmation, and FTDI disclaims all liability for system designs and for any applications assistance provided by FTDI. Use of FTDI devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify and hold harmless FTDI from any and all damages, claims, suits or expense resulting from such use. This document is subject to change without notice. No freedom to use patents or other intellectual property rights is implied by the publication of this document. Neither the whole nor any part of the information contained in, or the product described in this document, may be adapted or reproduced in any material or electronic form without the prior written consent of the copyright holder. Future Technology Devices International Ltd, Unit 1, 2 Seaward Place, Centurion Business Park, Glasgow G41 1HH, United Kingdom. Scotland Registered Company Number: SC136640