
TECHNICAL SPECIFICATION

Universal Monitor Interface

Purpose

This technical paper describes the *Universal Monitor Interface* (UMI in further text). It is designed to facilitate implementation of target monitors.

Revisions

Date	Version	Change
12.10.2006	9.7.27	Added position filed to ADDRESS specification.

Introduction

The Universal Monitor Interface (UMI) provides a framework to write monitoring software that covers a wide variety of targets and data storage technologies. It is designed with enough headroom to support new technologies as they become available.

Storage Devices

Today two types of storage devices are used in an embedded system. The classical approach maps the entire device in the CPU address space. Reading from such devices is no different than reading from RAM. Such devices are called ***Linear devices***. NOR FLASH is a typical representative.

Newer and larger devices like NAND FLASH do not map entirely in the CPU address space, but are accessible via dedicated bus interfaces like NAND port, SATA etc. Such devices are called ***Virtual devices***.

Storage Device Handling in winIDEA

winIDEA provides following device access features:

- Erase
- Program (Write)
- Verify
- Secure / Unsecure
- Display

All available features are accessible throughout the CPU debug session.

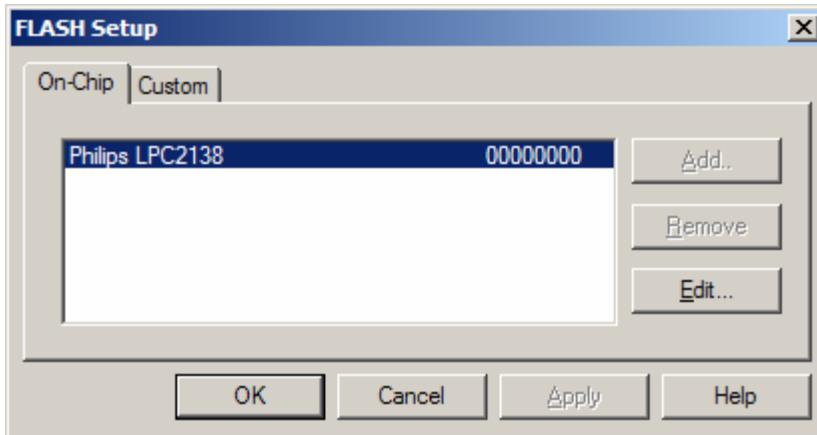
Device configuration

Any number of storage devices can be configured in winIDEA via ***Hardware/FLASH Setup*** dialog.

Through CPU selection some on-chip storage devices are implied. The ***On-Chip*** tab displays a list of integrated devices and provides means to configure the way winIDEA manipulates them.

External (to CPU packaging) storage devices are configured on the ***Custom*** tab.

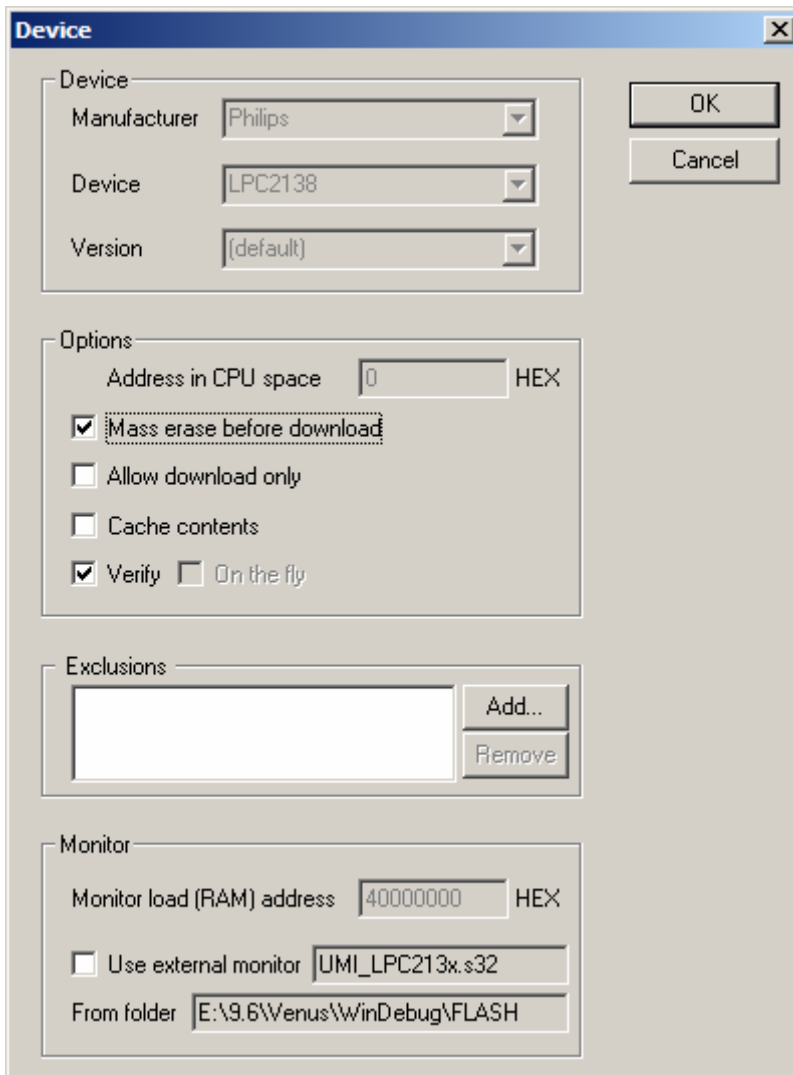
Note: winIDEA will allow configuration of any number of devices, but device overlapping in the address space is not allowed.



To add a custom device, click the **Add** button on the **Custom** tab. To remove it, click the **Remove** button.

Note: On-chip devices can not be added or removed.

Device options are configured by clicking the **Edit** button:



Device

The *Device* group provides storage device selection option. All options for in group are disabled for on-chip devices since the device is implied by CPU selection.

Options

Address in CPU space

defines the offset of the first device location within CPU address space. This option is disabled if the device is located on a fixed address or for *virtual* devices.

Mass erase before download

If this option is checked, a mass erase will be performed before program download. This option should be used if mass erase is a considerably faster operation compared to sequentially erasing the sectors affected by the download.

Allow download only

If this option is checked, no write operations into the device will be performed by winIDEA after the download completes. Use it to prevent usage of software breakpoints within the device as well as accidental change of device contents.

Cache contents

This option can considerably improve performance when device contents are changed during debug session (for example to set software breakpoints).

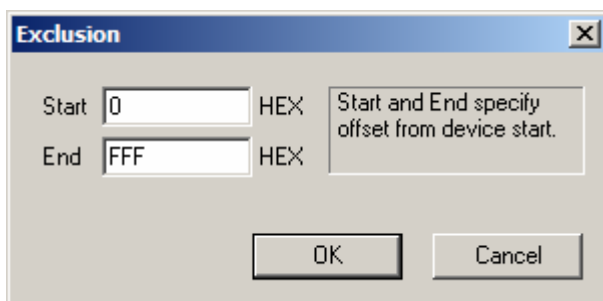
Note: use this option only if the target application does not modify the contents on its own.

Verify

If checked, every write operation will be verified. If the *On the fly* option is checked, the verification is performed by the UMI monitor. If the UMI monitor supports on-the-fly verification, usage of this option is recommended for performance reasons.

Exclusions

The *Exclusions* group allows specification of ranges to be excluded from programming. All ranges must be given using offset from device start.



Note: if writing to a device requires a part of the device to be erased, the excluded areas that are erased are programmed with values they held prior to the erase operation.

Monitor

The **Monitor** group allows adjustments to UMI monitor loading.

Monitor load (RAM) offset

If the monitor is relocatable, specify where it should be loaded. The specified address should denote RAM storage of sufficient size to hold the monitor.

Use external monitor

winIDEA comes with a range of supported device, for which the description and monitor files are included in the winIDEA binary. If a custom UMI monitor should be used for the selected device, this option should be checked and the UMI monitor file provided in the displayed location.

Note: Add-on devices with external descriptions always load the external monitor.

Device operations

Download

Invoking the **Debug/Download** operation will:

- Mass erase all devices with the **Mass erase before download** option set
- Program the code that falls in the device region. If the device was not mass erased and the programming can not be performed without erasing, the affected sectors are erased.

Note: Debug/Download only affects **linear** devices.

Display

Contents of **linear** devices can be viewed in regular memory windows. If write access to the device is not blocked with the **Allow download only** option, the device contents can be modified just like RAM locations.

Erase, Secure, Unsecure, Blank check

For every configured device a dedicated sub-menu is shown in the **Hardware** menu. The following operations are accessible from it:

- (mass) Erase
- Secure
- Unsecure
- Blank check

Note: only operations available on the device and provided by the monitor are listed.

UMI Specification

Device description

Storage device description is provided in a file with *.FLASH* extension.

FLASH file syntax

FLASH file uses standard ASCII text format. It contains the following fields:

Note: field parameters that contain whitespace characters must be enclosed in double quotes to be processed correctly.

HEADER

This must be the first field in the file. It identifies the device.

```
HEADER <manufacturer> <device> <description>
```

- **manufacturer**: the name of the manufacturer
- **device**: the name of the device
- **description**: description of the device

Example:

```
HEADER Samsung K9F5608 "HRP SC2410M iMO, S3C2410A, k9F5608, external monitor"
```

TYPE

Defines the monitor interface type. Currently only type **UNIVERSAL** is supported.

```
TYPE UNIVERSAL
```

ADDRESS

This field specifies the address of the device in the CPU address space. If this field is omitted, then the device is considered relocatable and its location must be specified in the device configuration dialog.

```
ADDRESS <address/hex> [position]
```

- **address**: address in CPU space.
- **position**: if this field is omitted, the device position is considered absolute
 - **R**: the device is relocatable.
 - **A**: the device is not relocatable.

Example:

```
ADDRESS 20000000
```

SIZE

Defines the size of the device. This field should be used for *linear* devices only. *Virtual* devices should use the **VIRTUAL** field.

```
SIZE <size/hex>
```

- **size**: size of the device.

Example:

```
SIZE 10000
```

VIRTUAL

This field indicates a virtual storage device. Such device is not mapped linearly into CPU memory space and uses a special interface to access its contents (NAND, SPI, etc.)

```
VIRTUAL <size/hex>
```

- **size**: size of the device.

Example:

```
VIRTUAL 1000000
```

EMPTY

Defines content of an erased device. If this field is omitted the default value of 0xFF is assumed.

```
EMPTY <value/hex>
```

- **value**: the value of an erased memory cell in hexadecimal format

Example:

```
EMPTY FF
```

TIMEOUT

Defines maximum time to wait for completion of any operation. If this field is omitted the default timeout of 1 second is used.

```
TIMEOUT <time/dec>
```

- **time**: number of seconds to wait

Example:

```
TIMEOUT 12
```

MONITOR

Specifies the name of the monitor executable and its load information. The executable must be in Motorola S format located in the same directory.

```
MONITOR <name> [R|A <monitor address/hex>]
```

- **name**: file name (with file extension) of the monitor executable. (Directory must not be given.)
- **R**: the monitor is relocatable and can be loaded to any address.
- **A**: the monitor is not relocatable (absolute) and must be loaded to **monitor address**.
- **Monitor address**: monitor load address in hexadecimal format. If the monitor is relocatable, this is the preferred load address.

Note: If **R**, **A** and **<monitor address>** parameters are omitted, winIDEA will extract the required information from the monitor header.

Example:

```
MONITOR S3C2410x.s32
```

UNIT

Defines an entity in the device that can be manipulated (erased, programmed) separately. This field is used for *linear* devices which map linearly into CPU memory space. For such devices at least one *UNIT* definition field must be specified. There can be multiple *UNIT* definition fields.

UNIT <offset/hex> <size/hex> <number of units/hex> <magic/hex> <entity size/hex> [flags/hex]

- **offset:** offset of this unit(s) from the start of the device.
- **size:** size of one unit.
- **number of units:** number of consecutive (sub) units of the same size.
- **magic:** the value that will be passed in the FLAGS/UNIT parameter OR-ed with the sub-unit number.
- **entity size:** minimum number of locations that can be written at once (usually 1 – 8).
- **flags:** the value that will be passed in the FLAGS parameter
 - **00000001:** partial writes supported – the monitor can reprogram the unit without prior erase

Example:

```
UNIT 00000000 1000 4 00030000 100 00000001
UNIT 00004000 4000 2 000F0400 100 00000000
UNIT 0000C000 1000 2 00030600 100 00000001
```

In this example following units are defined

ADDRESS	SIZE	UNIT	SUB-UNIT	SUB-UNIT MAGIC
0000.0000	1000	0	0	0003000 0
0000.1000	1000	0	1	0003000 1
0000.2000	1000	0	2	0003000 2
0000.3000	1000	0	3	0003000 3
0000.4000	4000	1	0	000F040 0
0000.8000	4000	1	1	000F040 1
0000.C000	1000	2	0	0003060 0
0000.D000	1000	2	1	0003060 1

If address 0x9000 is programmed, the magic value passed in FLAGS/UNIT field will be 000F0401.

Note: value of the *magic* is a contract between the *FLASH* file and the monitor. Other than OR-ing in the number of the sub-unit, winIDEA makes no use of this field. The field can be used to relieve the monitor of complex processing. In the above example, LSB byte 1 holds the number of sub-units before this unit, and LSB byte 2 the size of the unit in kB minus 1.

FLAGS

Defines up to four 32 bit values that are passed to the monitor upon every call. The purpose of this field is to allow usage of single monitor executable for multiple similar devices. If this field is omitted values of zero will be passed to the monitor.

FLAGS <flags0/hex[<flags1/hex[< flags2/hex>[< flags3/hex>]]>>>

- **flags0, flags1, flags2, flags3:** values passed to the monitor.

The following macros can be used in place of *flags1* – *flags3*

- UNIT the value of current UNIT's *magic* field ORed with the sub-unit number
- FREQ CPU frequency in kHz

Example:

```
FLAGS 00000001 UNIT FREQ
```

Note: *flags0* can not be used for *UNIT* or *FREQ* macros.

REGION

Defines a region of special properties within the device. There can be multiple *REGION* definition fields.

```
REGION <offset start/hex> <offset end/hex> <flags/hex>
```

- *offset start*: offset of region start from device start.
- *offset end*: offset of region end from device start.
- *flags*: property of the region.
 - **00000001**: excluded region – no write access allowed.
 - **00000002**: special value – after writing to this region, the new value will differ from the written value.

Example:

```
REGION 14 17 2
```

FEATURE

Defines a property of the device or monitor. There can be multiple *FEATURE* definition fields.

```
FEATURE <feature>
```

- *feature*:
 - **ON_THE_FLY_VERIFY**: on the fly verify available
 - **OVERWRITE_EMPTY**: only entities (see *UNIT*) which contain *EMPTY* value can be written.
 - **OVERWRITE_TOGGLE_EMPTY**: entities where the new value can be obtained by toggling the empty bits (typically programming a 00 over FF) can be programmed without prior erase.
 - **NO_DEVICE_ERASE**: mass device erase feature is not available. The device should be erased by sequentially erasing all *UNITs*.
 - **SECURE**: device secure operation is available.
 - **UNSECURE**: device un-secure operation is available.
 - **BLANK_CHECK**: device blank check operation is available.

Example:

```
FEATURE ON_THE_FLY_VERIFY
```

```
FEATURE OVERWRITE_TOGGLE_EMPTY
```

RESPONSE

Defines the message text to be displayed when the monitor returns an error code. There can be multiple *RESPONSE* definition fields. If no *RESPONSE* definition field is given, the return values will be shown in numeric format.

```
RESPONSE <value/hex> <message>
```

- *value*: value returned by the monitor. Bits 0 and 1 are reserved as well as value 0.
- *message*: the string to be displayed.

Example:

```
RESPONSE 04 "The FLASH clock divider was already initialized to a different value"
```

```
RESPONSE 08 "The device is secured"
```

Monitor binary

The monitor image is composed of:

- Header, containing:
 - Info block: used to provide information about the monitor itself
 - Parameter block: where the necessary information to direct the monitor is written by winIDEA.
 - Result and work area: here the return value of the monitor operation is stored. The work area serves as a region to store monitor's private data that must persist between calls.
- Code: the monitor executable code
- Data: the monitor global data.
- Stack: monitor application stack.
- Program buffer: the buffer where the data to be written/read/verified is placed.

Note: it is critical to keep the header, code, data, stack and program buffer contiguous. They can be allocated in any order, except for the header, which must be the first section.

Monitor Header

The monitor header is an array of 32-bit integers. The **EUnivMonHeader** enum specifies the indexes and their meaning. This and other definitions can be found in the UnivMon.h file.

```
enum EUnivMonHeader
{
    // 0000 Header
    umh0x12345678 = 0x00, // 0000 Endian identification
    umhHeaderFlags = 0x01, // 0004 Information about the monitor, use EHeaderFlags
    umhImageSize = 0x02, // 0008 Size of memory required by the monitor (code + data)
    umhEntryPoint = 0x03, // 000C monitor entry point
    umhExitPoint = 0x04, // 0010 exit point
    umhBufferAddress = 0x05, // 0014 absolute position of the buffer
    umhBufferSize = 0x06, // 0018 size of the program buffer
    umhWorkAreaSize = 0x07, // 001C size of Work Area
    umhFiller08 = 0x08, // 0020 SBZ
    umhWorkAreaAddress = 0x09, // 0024 absolute position of work area
    umhFiller0A = 0x0A, // 0028 SBZ
    umhFiller0B = 0x0B, // 002C SBZ
    // 0030 Parameters
    umhParameterFlags = 0x0C, // 0030 Information about the current call, use EParameterFlags
    umhAddress = 0x0D, // 0034 Address to program
    umhSize = 0x0E, // 0038 Number of bytes to program
    umhDeviceOffset = 0x0F, // 003C Device Offset in target memory space
    umhCustomFlags0 = 0x10, // 0040 Values specified in FLASH definition file
    umhCustomFlags1 = 0x11, // 0044 Values specified in FLASH definition file
    umhCustomFlags2 = 0x12, // 0048 Values specified in FLASH definition file
    umhCustomFlags3 = 0x13, // 004C Values specified in FLASH definition file
    // 0050 Result
    umhResult = 0x14, // 0050 Result of the operation
    umhExtResult = 0x15, // 0054 Ext.result - native error code of part's FLASH library

    umhNum = 0x16 // number of entries
};

enum EUnivMonLimits
{
    umlImageSize = 0xF000, // maximum total image size
    umlDownloadImageSize = 0x8000, // maximum download image size
    umlWorkAreaSize = 0x1000, // maximum work area size
};

enum EHeaderFlags
{
    hfRelocatable = 0x00000001, // monitor is relocatable
    hfVerify = 0x00000002, // on the fly verification is available
};
```

```

    hfSWBPOnExit      = 0x00000004,    // monitor already implements a SW BP instruction on
                                // Header.Exit_Point
    hfRequireLastCall = 0x00000008,    // call the monitor with pfLastCall before unloading
    hfRead            = 0x00000010,    // read functionality is available

    hfVerionMask      = 0xFF000000,    // these bits encode the version
    hfVersion0        = 0x00000000,    // version 0
};

enum EParameterFlags
{
    pfCallMask        = 0x00000003,
    pfFirstCall       = 0x00000001,    // this is the first call since the monitor was loaded
    pfLastCall        = 0x00000002,    // this is the Last call before the monitor is unloaded
    pfWrite            = 0x00000100,    // write the data
    pfVerify           = 0x00000200,    // verify
    pfErase            = 0x00000400,    // erase, or erase before writing
    pfProtect          = 0x00000800,    // protect the sector/device
    pfRead             = 0x00001000,    // read data
    pfDeviceScope      = 0x00004000,    // operation requested on entire device
    pfForce            = 0x00008000,    // try to recover from errors (ECC correction)
};

```

Valid **EParameterFlags** combinations:

```

pfWrite
pfWrite | pfVerify      - verify after write
pfWrite | pfErase       - erase before write
pfWrite | pfErase | pfVerify - erase, write, verify

pfErase
pfErase | pfVerify      - erase sector and blank check
pfErase | pfDeviceScope - mass erase
pfErase | pfVerify | pfDeviceScope - mass erase and blank check

pfProtect
pfProtect | pfVerify    - check sector protection
pfProtect | pfDeviceScope - protect device
pfProtect | pfDeviceScope | pfVerify - check device protection
pfProtect | pfForce     - unprotect sector
pfProtect | pfDeviceScope | pfForce - unprotect device

pfVerify
pfVerify | pfDeviceScope - blank check device

pfRead
pfRead | pfForce         - read with recovery if available

```

Monitor entry point

On entry, the monitor should initialize the stack and then call the code that processes the current parameters. Include the initialization steps required by the compiler CRT (BSS cleanup, exception handlers, etc.)

```

.global _MonitorEntryA
_MonitorEntryA:
    ldr sp, _Lstack_end /* set new stack */
    bl MonitorEntry     /* call MonitorEntry */
.global _MonitorExitLoopA
_MonitorExitLoopA:
    b _MonitorExitLoopA /* debugger will set a BP here */
_Lstack_end:
    .long __STACK_END__

```

How the monitor is used

In order to write data to a device, winIDEA loads the monitor into the target. The monitor capabilities are dynamically discovered by parsing the information contained in the monitor's header and the accompanying description file (*.FLASH* extension). To allow FLASH device access during debug session without corrupting the current application context, winIDEA preserves the entire application context before the monitor is loaded and restores it after the access is finished. While writable access is transparent to the user, the following actions are taken to perform the access:

1. Target application is stopped
2. Monitor image is loaded from disk into winIDEA's internal buffer
3. Monitor header is parsed and checked for consistency
4. Target application memory that will be used by the monitor is read and remembered. The Monitor Base Address (MBA) is defined by the S records in the monitor image file; the image size is defined by monitor header umhImageSize field.
5. All CPU core registers are read and remembered.
6. All CPU on-chip breakpoints are cleared.
7. Monitor image is loaded into target memory.
8. Requested operation parameters are loaded into umhParameterFlags through umhCustomFlags3 header fields.
9. Work Area (specified by umhWorkAreaAddress/umhWorkAreaSize) is loaded with the contents retrieved from the previous monitor invocation. On first ever invocation, the work area is zeroed.
10. Program buffer (specified by umhBufferAddress/umhBufferSize) is loaded with the data to be programmed.
11. If umhHeaderFlags does not set hfSWBPOnExit flag, a hardware breakpoint is set to umhExitPoint.
12. Program counter is preset to umhEntryPoint.
13. CPU is released to running.
14. winIDEA polls for stopped state, waiting for monitor to hit the exit point. If within specified time (specified in the FLASH file, TIMEOUT field) the exit point is not reached, winIDEA proclaims a monitor timeout.
15. Work Area is read and remembered for the next invocation.
16. Memory remembered in step 4 is written to the target.
17. On-chip breakpoints cleared in step 6 are restored.
18. CPU core registers remembered in step 5 are restored.

winIDEA also sets properly the parameter block and loads the data buffer with the data to be written.

Note: If any other target resources are used by the monitor, it is the responsibility of the monitor to restore them before exiting.

The monitor code

The full code for this example is available in the UMI samples folder. Here we only show how the monitor header is constructed and discuss a special non-standard feature of the GNU C compiler that controls section assignment: attributes. Note that different compilers provide this capability by different means (commonly #pragma directives).

```
#include "UnivMon.h"

extern _MonitorEntryA();
extern _MonitorExitLoopA();
extern char __SIZE_IMAGE__; /* fake symbol calculated at link time */

// device specific definitions
#define DEVICE_SIZE 0x100000
#define SECTOR_SIZE 0x1000

/* program buffer; data that is to be transferred to the device */
#define PROGRAM_BUFFER_SIZE SECTOR_SIZE
BYTE g_abyProgramBuffer [PROGRAM_BUFFER_SIZE] __attribute__((section (".progbuf")));

/* work buffer */
struct SWorkBuffer
{
    BYTE m_abySectorProtected[DEVICE_SIZE/SECTOR_SIZE/8];
    BOOL m_bDeviceProtected;
} g_WorkBuffer __attribute__((section (".work")));

/* Header */
struct SMonParam g_MonParam __attribute__((section (".fmonpar"))) =
{
    0x12345678, /* Endianness mask */
    hfVerify | hfVersion0, /* Header flags: not relocatable, on the fly verify
available, Spec. Version 0 */
    (DWORD) &__SIZE_IMAGE__, /* Image size */
    (DWORD) &_MonitorEntryA, /* Entry point */
    (DWORD) &_MonitorExitLoopA, /* Exit point */
    (DWORD) &g_abyProgramBuffer, /* Buffer address */
    PROGRAM_BUFFER_SIZE, /* Buffer size */
    sizeof g_WorkBuffer, /* Work area size */
    0, /* Reserved 8 */
    (DWORD) &g_WorkBuffer, /* Work Buffer Address */
    0, /* Reserved A */
    0, /* Reserved B */

    /* Parameters */
    0, /* Parameter flags */
    0, /* Address */
    0, /* Size */
    0, /* Reserved 4 */
    0, /* Custom flags 0 */
    0, /* Custom flags 1 */
    0, /* Custom flags 2 */
    0, /* Custom flags 3 */

    /* Result */
    fpcOK, /* Result */
    0, /* Reserved 5 */
};
```

Note how the data buffer and the header declarations are followed by an “attribute” modifier. The parameter to this attributes (“section”) tells the compiler where to put the related data. This combined with the linker command file allows creating images where you can control the exact location of your program image parts. For information about these please refer to your compiler’s documentation.

The **MonitorEntry** function demonstrates how to extract parameters and execute requested actions.

```
#define FLAG(F) (0 != (g_MonParam.m_dwParameterFlags & F))
void MonitorEntry()
{
    if (FLAG(pfFirstCall)) {};

    BOOL bDevice = FLAG(pfDeviceScope);
    BOOL bErase = FLAG(pfErase);
    BOOL bVerify = FLAG(pfVerify);
    BOOL bForce = FLAG(pfForce);
    DWORD dwSectorAddress = g_MonParam.m_dwAddress & ~(SECTOR_SIZE - 1);

    g_MonParam.m_dwResult = fpcOK;
    if (FLAG(pfWrite))
        Monitor_Write(bErase, bVerify);
    else if (bErase)
        Monitor_Erase(bDevice, dwSectorAddress, bVerify);
    else if (FLAG(pfProtect))
        Monitor_Protect(bDevice, dwSectorAddress, !bForce, bVerify);
    else if (bVerify)
        Monitor_BlankCheck(bDevice, dwSectorAddress);
    else if (FLAG(pfRead))
        Monitor_Read(bForce);
    else
        g_MonParam.m_dwResult = fpcFAIL | fpcNotImplemented;

    if (FLAG(pfLastCall)) {};
}
```

To access custom flags passed to the monitor (see *FLASH File Syntax/FLAGS*) use:

```
DWORD dwFlags0 = g_MonParam.m_dwCustomFlags0;
DWORD dwUNITMagic = g_MonParam.m_dwCustomFlags1;
DWORD dwFrequency = g_MonParam.m_dwCustomFlags2;
```

The linker command file

The below excerpt shows how different sections must be linked to insure required monitor image layout.

```
__BASE__ = 0x40000000; /* image start */
__STACK_SIZE__ = 0x100; /* stack size */

SECTIONS
{
    /* Universal Monitor Specification image header */
    .fmonpar __BASE__ :
    {
        *(.fmonpar)
    }

    /* code */
    . = ALIGN(4);
    .text : { *(.text) }

    /* read only data */
    . = ALIGN(4);
    .rodata : { *(.rodata) }

    /* working area preserved between monitor calls */
    /* stick it behind text & rodata so it can be loaded together with them */
    . = ALIGN(4);
    .work (NOLOAD) :
    {
        *(.work)
    }

    /* initialized data */
```

```

    . = ALIGN(4);
    .data : { *(.data) }

    /* noninitialized data */
    . = ALIGN(4);
    .bss :
    {
        *(.bss)
        *(COMMON)
        . = ALIGN(4);
    }

    /* program data buffer */
    . = ALIGN(4);
    .progbuf (NOLOAD) :
    {
        *(.progbuf)
    }

    /* stack */
    . = ALIGN(4);
    __STACK_END__ = . + __STACK_SIZE__;

    /* define the __sizeImage symbol, so the entire monitor image size can be reported in the
    monitor header */
    __sizeImage = __STACK_END__ - __BASE__;
}

```

Debugging the monitor

The easiest way to debug the monitor is to write an alternate entry point which is used during a normal debug session. This entry point then calls another function – **main** is a good name choice.

```

/* MonitorA.s */
.code 32

/* This is the monitor entry point when called by winIDEA */
.global _MonitorEntryA
_MonitorEntryA:
    ldr sp, _Lstack_end    /* set new stack */
    bl MonitorEntry        /* call MonitorEntry */
.global _MonitorExitLoopA
_MonitorExitLoopA:
    b _MonitorExitLoopA    /* debugger will set a BP here */

/* This is the entry point for monitor development */
.global _MonitorEntryMain
_MonitorEntryMain:
    ldr sp, _Lstack_end    /* set new stack */
    bl main                 /* call main */
_MonitorExitLoopMainA:
    b _MonitorExitLoopMainA /* loop in case main returns */

_Lstack_end:
    .long __STACK_END__
.end

```

main ‘manually’ configures the parameter area and then calls **MonitorEntry**.

```

// for development purpose only. main isn't called in monitor operation
int main()
{
    // set all protections off
    int I;
    g_WorkBuffer.m_bDeviceProtected = FALSE;
    for (I = 0; I < sizeof g_WorkBuffer.m_abySectorProtected; I++)
        g_WorkBuffer.m_abySectorProtected[I] = 0;
}

```

```

// Parameters
g_MonParam.m_dwDeviceOffset = 0x0;
g_MonParam.m_dwAddress      = 0x3000;
g_MonParam.m_dwSize        = 0;

g_MonParam.m_dwParameterFlags = pfProtect;
MonitorEntry(); // Call the monitor entry point

g_MonParam.m_dwParameterFlags = pfProtect | pfVerify;
MonitorEntry();

g_MonParam.m_dwParameterFlags = pfProtect | pfForce;
MonitorEntry();
return 0;
}

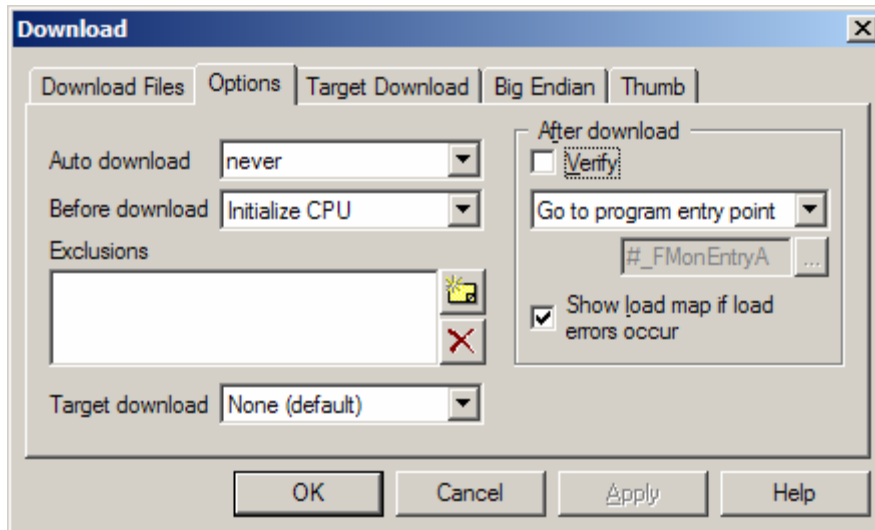
```

Add an **ENTRY** field in the linker command file, and configure winIDEA to 'Go to program entry point' after download.

```

/* Monitor.ind */
. . .
ENTRY(_MonitorEntryMain) /* debugger will use this as entry point */
. . .

```



When you perform a download now, execution will begin on `_MonitorEntryMain`, from where you'll be able to step through the monitor.

Adding the new device to winIDEA list

Custom FLASH devices are recognized by winIDEA when the `.FLASH` file and the monitor file are located in the `FLASH` subdirectory of winIDEA installation directory.

Custom devices will be shown with an asterisk following their name.

Testing the new device

Use operations described in the *Device Operations* section. In case you experience error conditions or timeouts, revert to technique described in the *Debugging the monitor* section.

Monitor writing tips and pitfalls

Examples

Several examples of UMI monitors are provided with winIDEA sample projects. They all operate with the supplied GNU compilers.

It is recommended that you stick with this compiler if available for the CPU architecture you use. Otherwise use the supplied configuration examples as a guide to porting them to your compiler.

Performance considerations

When FLASH programming is required, winIDEA will amongst others, take the following actions

- Preserve target memory required for the whole FLASH image, by uploading it and storing it
- Load the monitor. This includes the header, the executable code, constant data and the work area.
- Load the data to be programmed and adjust the header parameters.
- Execute the monitor
- Repeat the above two steps until all the data is programmed
- Restore target memory image and the CPU register context.

Depending on the debugging interface, extracting and loading memory incurs certain performance overhead. To minimize it, you should:

- Keep the monitor as small as possible. If possible, use no runtime libraries.
- Keep monitor parts that must be loaded (header, code and work area) contiguous.

NAND flash: large blocks and small pages

NAND flash chips are organized in blocks, pages and words; where a block is the fundamental erase unit, and a page is the write/read unit. Blocks usually contain many pages. This requires that in order to modify a page that is part of an already initialized block; you need rewrite all pages of the block. The easiest way to work around this characteristic is to make sure the device is erased before writing. This is how the examples are implemented. You can also write a monitor that takes care of reading the whole erase unit, modify it in memory and write it back.

Read operations on virtual devices

Virtual devices are devices that are not directly mapped into the processor's address space. These devices are commonly accessed by writing commands to control ports and reading writing the data from an access point. When winIDEA needs to read this kind of device it will call the monitor with the Read parameter flag set. The monitor is expected to fill the buffer with the requested data from the device.

Verify operations on virtual devices

On *linear* devices verify can be performed by winIDEA or the monitor (if the *Verify on the fly* option is set). On *virtual* devices verify can only be performed by the monitor. The monitor receives the buffer to verify on input and it should take care of comparing the data on the device to the buffer. The monitor should return a success error code when there are no differences. If a difference is found the monitor should return an error code.