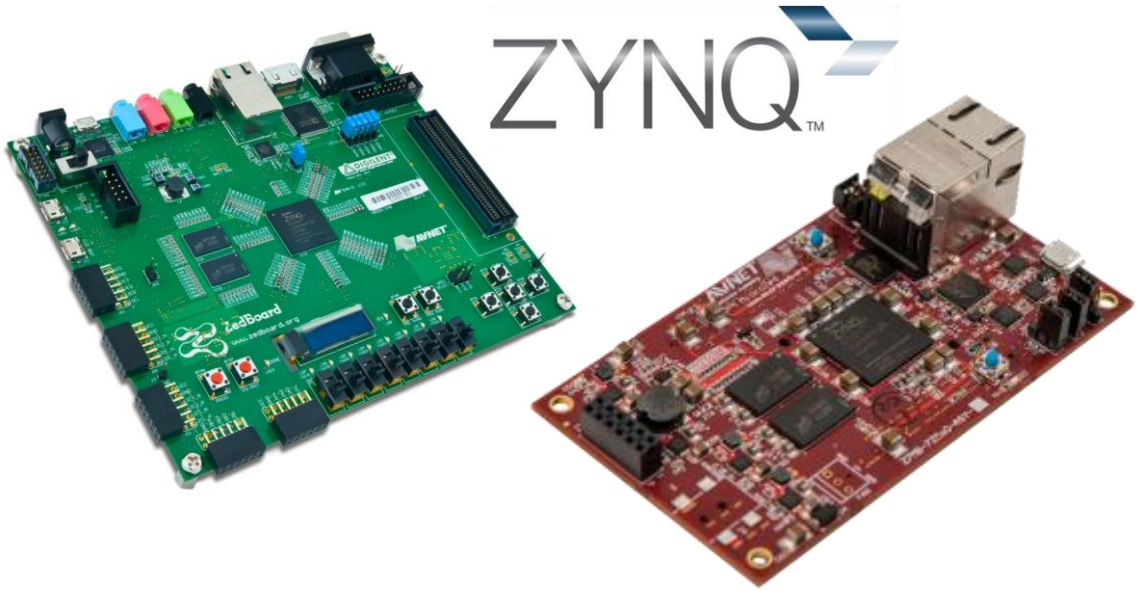



# Controlling the Zynq MMU / caches using a custom Xilinx SDK library

Version 1.0 - Vivado 2014.2

Rich Griffin
Xilinx Embedded Specialist, Silica EMEA

## Introduction

The default settings provided by Xilinx in the SDK for the Zynq devices and Cortex A9 processors will be sufficient for a large number of designs to function perfectly without any additional intervention. However it is often desirable for the user to take greater control of the settings on the processors to either boost performance in their designs, or to take advantage of specific features. One such example of a details processor setting is the control of the caches and Memory Management Unit (MMU). This guide shows how to control the various cache / MMU features using run time C code, and also explores how to integrate that functionality into a custom Xilinx SDK library for easier access and to facilitate design re-use.

## The Basics

Just like many application processor units, the Cortex A9 has some highly advanced features which enable users to achieve higher performance from their embedded products. Specifically, the Zynq 7000 series SoC devices are equipped with Level 1 and Level 2 caches, and an advanced Memory Management Unit (MMU). Caches are arrays of local memory situated close to the processor core which keep a local copy of smaller sections of main (external) memory. Accesses to external DDR memory in all processor systems can be slow and require many cycles of latency for each fetch and store instruction. Having a local copy of instructions and user data enables the processor to execute code much faster, by removing the need to access external devices for every executed processor instruction. The cache controllers use sophisticated algorithms to automatically fetch and locally store regions of the external DDR memory which are likely to be needed by the processor. Greater control of these algorithms is often desirable, for example if the user wants to control the caches to devote half of the cache array to a specific processor in a multi-core system, or to always keep a certain region of memory cached for better performance in specific parts of their design. Caching modes can also be adjusted to use "write back" and "write through" schemes, providing users with finer grain control of when external memory is updated by the cache controller. Sometimes it is also desirable to disable caching completely for certain regions of memory. Beyond the features directly attributable to caching, the MMU allows memory mapping features to be set, resulting in the translation of one memory address to another. In addition, it is possible to "flag" certain areas of memory to have specific behaviours such as "sharable", "read only", or even to be protected against unintentional execution. Examples of these latter behaviours can be attractive to programmers who have a security focus and, for example, can be extremely powerful ways to deny hackers the ability to execute rogue code from the stack which might have been placed there by way of a maliciously crafted buffer overflow attack.

No matter what the desired usage model may be, all of these settings are controlled by the various registers and configuration tables which are buried deep within the processor core. Accessing these advanced features can be a complex task requiring an intimate understanding of the underlying hardware architecture, and usually requires manual decoding of some very unintuitive hexadecimal register values. For this reason it is often desirable to write some function calls to automate the process.

The hard work can be done once by the user, and then forever more consigned to the inner workings of a carefully written software function.

## The Objective

It would be futile and foolish for any application note or technical guide to attempt to discuss every possible combination of features available to the user. The number of possible combinations are too vast to cover, and for those reasons this guide will focus on one specific goal and explain how to achieve it. If users wish to access other similar features, this guide can be used as a reference point allowing the reader to modify the flow to achieve their goals. A common request from users is the ability to enable or disable the processor caches for specific regions of memory and, less frequently, to control the caching mode from a write-through / write-back perspective. The objective of this guide will therefore be to create some custom software functions which will allow the user to control these features quickly and easily from a user application. To enable re-use and ease of use of the solution, this guide will also discuss how to create a custom Xilinx SDK library which will enable the new functionality to subsequently be added to any standalone board support packages with just a few clicks of the mouse.

## The Hard Theory

Before any code can be written we must first explore the technical documentation for the processor cores within the Zynq 7000 series devices, and use it to clearly understand how the cache controllers and MMUs operate. The first port of call is the Xilinx Zynq Technical Reference Manual (UG585); Users would be forgiven for assuming that caching modes would be discussed in the cache controller section of the documentation, however this is not the case. An example of this can be seen in chapter 3.4, which discusses the operation of the Level 2 cache and describes the various write-back / write-allocate / write-through modes. Even though the L2 cache controller supports all of the functionality that we wish to control, there is no guidance on how these features are controlled from the cache controller. Further investigation reveals that this is not an oversight but is correct behaviour because the L2 cache controller simply accepts requests from the memory management unit, and processes them according to the type of memory access that has been requested. Although the L2 Cache controller can be configured to selectively lock the cache arrays to a specific CPU core, the control of the caching modes (i.e. for cacheable / non-cacheable / write-back / write-through) are determined by the memory management unit. Our focus must therefore move to chapter 3.2.5 of the Technical Reference Manual, which discusses the various features that are available and explains that they are all controlled by configuration entries which are located in "translation tables" in a known area of main memory. The term translation table comes from the previously discussed capability for the MMU to map (translate) virtual memory addresses to physical memory addresses. This functionality is often seen when an operating system is used; each user application is compiled and linked with the assumption that it is the only task running on the processor, whereas in reality multiple software applications are running and are given access to the

processor's execution time by way of scheduler that is built into the OS kernel. A critical point of understanding is that multiple software applications try to reference memory at the same address locations because they are all compiled and linked in the same way, and have no knowledge of each other. If no memory translation were to take place in the processor, an application could potentially interfere with another because they all reference the same addresses in memory. The MMU solves this problem by fooling the applications into believing that they are accessing the memory addresses that they request, but then translate the addresses in such a way that each user application has its address space mapped to a different location in physical memory. This explains the basic concepts of virtual memory (which is what the user application sees), and physical memory (which is what the processor hardware sees). In the example of an operating system, these mappings of memory space from the virtual to physical addresses may need to constantly change as new applications are loaded and closed, so the translation tables are held in a "soft" format in main memory. This dedicated "soft" region of memory is known as the translation table or page table, and is usually controlled by the operating system's kernel.

Now that the basic theory of address translation is understood, it is also important to consider that not all user applications require the same amount of memory in order to execute. The mapping of virtual to physical addresses must occur with enough flexibility to cater for both small and large applications, but without wasting physical memory in the embedded system. To achieve this, Memory Management Units usually permit various different sizes of memory regions to be allocated for address translation, and these regions are often known as "sections" or "pages". To offer maximum flexibility, the MMU in the Zynq 7000 series Cortex A9 processor allows four different sizes of these regions to be controlled. These are discussed in the Technical Reference Manual and are known as 4KB and 64 KB "pages", 1MB "sections", and 16MB "super-sections". The translation table / page table is therefore simply a list of the various memory regions, details of their virtual to physical address mappings, and whether they have any special attributes or behaviours such as "cacheable", "non-cacheable", "sharable", "read-only", or not executable (known as "execute never"). There are trade-offs to be made here; smaller pages give the user finer granularity of memory, and therefore need more memory to be devoted to a larger page table, whereas larger sections or super-sections offer coarse granularity of memory but the amount of memory required to store the page tables is smaller. For very fine grain control of memory, a concept exists whereby two different pages tables are used in unison; Level 1 and Level 2 page tables. Level 1 page tables (coarse grain) can point to either large sections of memory, or optionally to entries in a Level 2 page table (fine grain). This latter usage of Level 2 page tables allows very detailed and fine grain control of the memory, but this is beyond the scope of this guide. It should be noted that even this lengthy description is still a very simplistic view of the operation and capabilities of the MMU, but it is adequate for the purposes of this guide.

For our purposes, we will be keeping things relatively simple by defining the behaviour of memory which we shall divide into 1MB sections. A Level 1 page table is sufficient to achieve this level of granularity, and therefore we shall be concerning ourselves with a page table that has 4096 entries, which in turn gives control over

the entire 4GB address space (4096 * 1MB). The full details of the Level 1 page tables can be found in Figure 3-5 of the Zynq Technical Reference Manual. Level 1 page tables can contain four different types of page table entries, each type being denoted by the binary encodings in bits 1:0 of the table entry, but we are concerned with the 1MB "section" entry which is highlighted below.

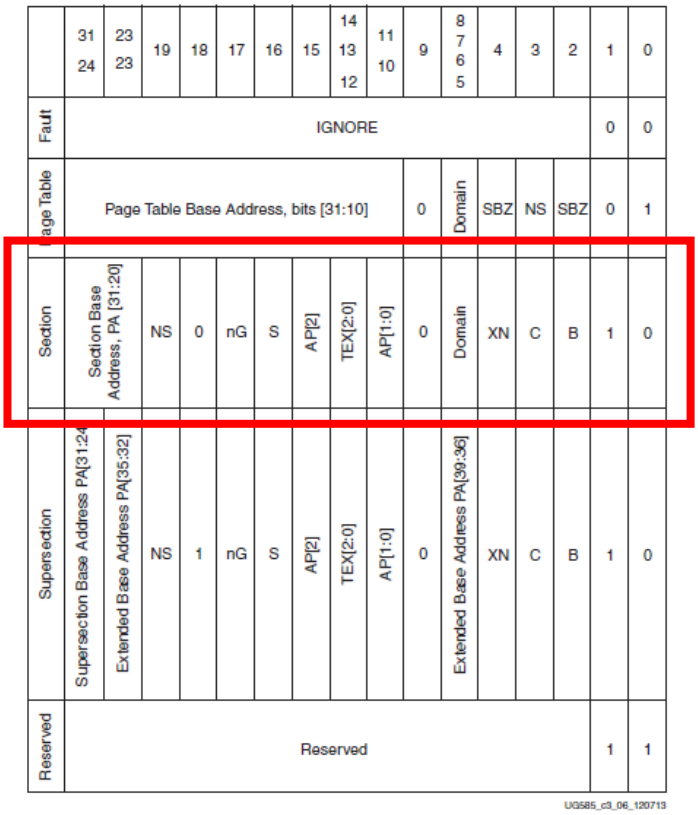



*Figure 3-5:* **L1 Page Table Entry Format**

The first key point of understanding to learn from this table is that the highlighted area shows one of 4096 similar entries which are all described in the Level 1 page table. The location of the entry in the page table denotes which area of physical memory it controls. The first entry in the table therefore controls the behaviour of the first 1MB section of memory at physical addresses 0x00000000 → 0x000FFFFF, the second entry in the table controls the second 1MB section at physical addresses 0x00100000 → 0x001FFFFF, and so on until the last table entry (i.e. the 4096$^{th}$) which controls physical addresses 0xFFF00000 → 0xFFFFFFFF. In the Zynq 7000 series devices we have a DDR memory controller which covers 1GB of memory space and is located at addresses 0x00000000 → 0x3FFFFFFF *(Note: The OCM can also optionally be mapped to some of this memory region, but this is a topic beyond the scope of this guide)*. Therefore the first 1024 entries in the Level 1 page table control how the DDR behaves (1024 * 1MB).

Each entry in the Level 1 page table has a number of attributes which we must understand in order to control the behaviour of the memory in that section. The upper 12 bits of the entry (bits 31:20) control the base address of the 1MB virtual address region that maps into the physical address space. We will keep things simple by not introducing any address translation for the purposes of this guide (i.e.

virtual addresses will be the same as physical addresses). Therefore the upper 12 bits of the table entry are always the same as the 12 bits of the offset of the page table entry's address (i.e. 0x000000 for the first entry, 0x100000 for the second, 0x200000 for the third, and so on). Bits 1:0 of each page table entry will always be "10" because this is the coding which tells the MMU that the page table entry refers to either a 1MB "section" or 16MB "super-section". Bit 18 of the entry is always "0" because that denotes that the table entry is for a 1MB "section" rather than a 16MB "super-section" (super-sections would be denoted by a "1" in this bit). The remaining bits control how the memory in that region behaves, and are explained in the Technical Reference Manual in the sub-chapter entitled "Description of Page Table Entry Fields". This behaviour is summarised briefly below.

| Bit Name | Description |
|---|---|
| **NS** (Bit 19) | Always set to "0". |
| **nG** (Bit 17) | Non Global bit. Denotes that this entry in the page table only applies to the associated "ASID" (Address Space IDentifier). This is an advanced feature used by operating systems and allows the OS to create a page table entry that is only specific to one task that is running under the OS. This topic is beyond the scope of this guide, but we will leave the settings as "global"; thus nG=0. |
| **S** (Bit 16) | Sharable bit. Denotes whether the memory can be shared by more than one master (i.e. CPU). We will be allocating memory as sharable, thus S=1. |
| **AP** (Bit 15 and Bits 11:10) | Access Permission bits. Denotes whether the memory is Full access, read only, privileged access, or completely inaccessible by the CPU. See table 3-2 of the TRM. This topic is beyond the scope of this guide, but we will be assigning "Full Access" permissions to the DDR, thus AP[2:0] = "011". |
| **TEX** (Bits 14:12) | These bits are used in combination with the C & B bits, and denote the behaviour of "ordered" memory, including the caching modes. See Table 3-3 and 3-4 of the TRM, discussed below. |
| **Domain** (Bits 8:5) | Controls to which of 16 possible domains the page table entry belongs. Domains can be used to control the behaviour of a number of page table entries in a single group. Permissions assigned to a domain then apply to all page table entries assigned within that domain. This topic is beyond the scope of this guide, but we will be assigning all of the page table entries for the DDR to the "1111" domain. |
| **XN** (Bit 4) | Execute Never bit. This denotes that the processor cannot execute from the section of memory described by the page table entry. This can be a useful feature when writing software with a security focus, for example to prevent execution from certain areas of memory which could be targeted by hackers by way of a buffer overflow attack. We will be permitting execution; thus XN=0. |
| **C** and **B** (Bits 3:2) | These bits control the caching mode, and are used in combination with the TEX bits. See Table 3-3 and 3-4 of the TRM, and also discussed in detail below. |

Our goal is to control the caching mode of the DDR memory in various 1MB regions of the memory map. Using this analysis of the Level 1 page table entries, we can see that our focus must be on the "TEX", "C", and "B" bits. Table 3-3 and Table 3-4 from the TRM provide us with further guidance on the precise settings, and are reproduced below for clarity.

*Table 3-3:* **Memory Attributes Encodings**

| TEX [2:0] | C | B | Description | Memory Type |
|---|---|---|---|---|
| 0 | 0 | 0 | Strongly-ordered | Strongly ordered |
| 0 | 0 | 1 | shareable device | Device |
| 0 | 1 | 0 | Outer and Inner write through, no allocate on write | Normal |
| 0 | 1 | 1 | Outer and Inner write back, no allocate on write | Normal |
| 1 | 0 | 0 | Outer and Inner non-cacheable | Normal |
| 1 | - | - | Reserved | - |
| 10 | 1 | 0 | Non-Shareable device | Device |
| 10 | - | - | Reserved | - |
| 11 | - | - | Reserved | - |
| 1XX | Y | Y | Cached memory<br>XX – Outer Policy<br>YY – Inner Policy | Normal |

*Table 3-4:* **Memory Attributes Encodings**

| Encoding Bits | | Cache Attribute |
|---|---|---|
| C | B | |
| 0 | 0 | Non-cacheable |
| 0 | 1 | Write-back, write-allocate |
| 1 | 0 | Write-through, no write-allocate |
| 1 | 1 | Write-back, no write-allocate |

The DDR memory is regarded as being "Normal" memory type, and we are trying to control just the caching modes. We shall therefore be using the combination of bits described on the last row of Table 3-3 where TEX(2) = 1, and TEX(1:0) are always set to match the C & B bits described in Table 3-4. Note that although TEX(1:0) is derived from the values of C & B, we must also set the values of C & B in their own bit positions.

Our required encoding for the TEX, C, and B bits are therefore as follows:

| Caching mode | TEX[2] | TEX[1] | TEX[0] | C | B |
|---|---|---|---|---|---|
| Non-cacheable | 1 | 0 | 0 | 0 | 0 |
| Write-back, write-allocate | 1 | 0 | 1 | 0 | 1 |
| Write-through, no write-allocate | 1 | 1 | 0 | 1 | 0 |
| Write-back, no write-allocate | 1 | 1 | 1 | 1 | 1 |

So by way of an example; let's imagine that we want to create a page entry which controls the first 1MB of the DDR memory space (i.e. the first page table entry, controlling physical addresses 0x00000000 → 0x000FFFFF).

For this example we will assume the following settings:

- Create a page table entry for a 1MB "Section"
- Global scope (nG=0)
- Sharable memory (S=1)
- Full Access Permissions (AP[2:0] = "011")
- Write-back / Write-allocate caching mode (C=0, B=1, TEX[2:0] ="101")
- Domain = "1111"
- Execution is permitted (XN=0)

| 31:24 | 23:20 | 19 | 18 | 17 | 16 | 15 | 14:12 | 11:10 | 9 | 8:5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Section Base Address (PA) [31:20] | | NS | 0 | nG | S | AP[2] | TEX[2:0] | AP[1:0] | 0 | Domain | XN | C | B | 1 | 0 |
| **0x000000** | | **0** | **0** | **0** | **1** | **0** | **101** | **11** | **0** | **1111** | **0** | **0** | **1** | **1** | **0** |

So the value that we must write to the first entry in the page table is **0x15DE6**.

By now it is probably very clear why this is an undesirable process to perform manually each time! Having a software function to automate these calculations would be of huge benefit to software developers, will reduce the possibilities of human error, and will save a lot of development time.

## Let's Write Some Software

The starting point of almost all software design is to assess what has already been written by somebody else. The world is arguably full of software code, and there is seldom any point in "re-inventing the wheel" unless there's an extremely good reason to do so. In the case of our example we are writing some additional C code to compliment the Standalone / Bare-metal Board Support Package (BSP) that is supplied with the Xilinx SDK tools. A quick look at the Xilinx BSP reveals that there is a supplied header file for MMU operations called "xil_mmu.h". Looking inside that header file we find a function which is designed to update the MMU page tables:

```
void Xil_SetTlbAttributes(u32 addr, u32 attrib);
```

The "TLB" part of the function name is an abbreviation for "Translation Look-Aside Buffer". We have already discussed the purpose of the page tables which list how each region of memory behaves and also describes the mapping between the virtual physical address space. One of the functions of the MMU hardware is to keep a local copy of the most recent translation table look-up operations (and thus avoiding additional external memory accesses), and so the purpose of the TLB is essentially a cache of the page tables. When the required translation for a memory region is not already present in the TLB, the MMU accesses the page tables which are stored in memory to fetch the page table entry that is relevant to the memory region. This is known as "page table walking". Setting a TLB attribute using this function is therefore a way of updating not just the page table, but also the cached TLB entry so that it can be immediately used by the software. In actual fact, the function updates the cached version of the page table in the TLB, and then flushes the caches to write the new page table back to external DDR.

We can see from the function prototype that two variables must be passed in to the function; the address of the memory region, and the attributes that are to be assigned. The "addr" field is easy, and simply represents the first address in the 1MB region of memory. The "attrib" field is the hexadecimal values that we calculated in our previous example. So in summary, there is a supplied function to update the MMU page tables / TLB, but not one which makes the process intuitive or easy to use.

Our goal is therefore to write a function which makes life easier for the software developer, by removing the need to manually calculate the hexadecimal value that will be passed into the existing function. One way of doing this is to create a series of enumerated #define values which represent each of the bit values in the page table entry. For example, we saw previously that the "sharable" bit (Bit 16) can be set when we want to enable that behaviour for a region of memory. The assertion of just bit 16 in a 32 bit word has a hexadecimal equivalent value of 0x00010000. We can therefore assign an intuitive enumerated name to this bit by creating a macro in C using the #define syntax.

```
#define SHAREABLE 0x10000  // S = b1
```

Theoretically, we can now set the "sharable" bit for an MMU page table by passing that macro into the supplied function from the BSP. So for the first 1MB region of DDR memory in a Zynq-7000 device, we could use:

```
Xil_SetTlbAttributes(0x00000000, SHARABLE);
```

Theory is fine, but this is clearly not going to work in practice because we have not set the rest of the bits in the page table entry to valid values. In fact we have cleared all of the other bits, and asserting just Bit 16 is not going to get us anywhere on its own! So to make this work we need to assign macros to the remainder of the bits in the page table entry. Some of them are easy (e.g. when the macro refers to a single bit, as shown above) but some of them must be combinations of different bits. For example, we previously saw that to enable Write-back / Write-allocate caching mode, we should set the "C" bit to 0, set the "B" bit to 1, and set the TEX[1:0] field to "01" (Note: TEX[2] = 1 for "Normal" memory types, so we needn't ever change it. We will therefore worry about controlling this bit later on). The combination of these bits can be represented in the same way as we did before, and a macro can be created:

| 31:24 | 23:20 | 19 | 18 | 17 | 16 | 15 | 14:12 | 11:10 | 9 | 8:5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Section Base Address (PA) [31:20] | | NS | 0 | nG | S | AP[2] | TEX[2:0] | AP[1:0] | 0 | Domain | XN | C | B | 1 | 0 |
| - | | - | - | - | - | - | **-01** | - | - | - | - | **0** | **1** | - | - |

So the hexadecimal value which represents this caching mode is **0x1004**.

```
#define WRITEBACK_WRITEALLOCATE  0x1004  // TEX(1:0) = b01, C = b0, B = b1
```

Using an identical process, we can assign a complete set of macros for the rest of the settings on the MMU. The following list also includes macros to represent some of the MMU access permission bits, although we did not discuss these in detail.

```
#define NON_CACHEABLE 0x00  // TEX(1:0) = b00, C = b0, B = b0
#define WRITEBACK_WRITEALLOCATE  0x1004  // TEX(1:0) = b01, C = b0, B = b1
#define WRITETHROUGH_NO_WRITEALLOCATE 0x2008  // TEX(1:0) = b10, C = b1, B = b0
#define WRITEBACK_NO_WRITEALLOCATE 0x300C  // TEX(1:0) = b11, C = b1, B = b1
#define NON_GLOBAL 0x20000  // nG = b1
#define EXECUTE_NEVER 0x10  // XN = b1
#define SHAREABLE 0x10000  // S = b1
#define AP_PERMISSIONFAULT 0x00  // AP(2) = b0, AP(1:0) = b00
#define AP_PRIVIEGED_ACCESS_ONLY 0x400  // AP(2) = b0, AP(1:0) = b01
#define AP_NO_USERMODE_WRITE 0x800  // AP(2) = b0, AP(1:0) = b10
#define AP_FULL_ACCESS 0xC00  // AP(2) = b0, AP(1:0) = b11
#define AP_PRIVILEGED_READ_ONLY 0x8800  // AP(2) = b1, AP(1:0) = b10
```

These macros are usually added to a header file for easy re-use. In our case we will create a header file called "mmu_control.h", and add it to the source files in the Xilinx SDK software application project.

The last part of the puzzle is to also calculate a hexadecimal value for the bits in the page table entries which never change. We saw previously that the "NS" bit is always 0, TEX[2] is always 1, Bit 18 and Bit 9 are fixed at 0, the "Domain" bits are always set to "1111", and Bits [1:0] representing the page table entry type are always set to "10".

| 31:24 | 23:20 | 19 | 18 | 17 | 16 | 15 | 14:12 | 11:10 | 9 | 8:5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Section Base Address (PA) [31:20] | | NS | 0 | nG | S | AP[2] | TEX[2:0] | AP[1:0] | 0 | Domain | XN | C | B | 1 | 0 |
| - | | **0** | **0** | - | - | - | **1--** | - | **0** | **1111** | - | - | - | **1** | **0** |

So the hexadecimal value representing the fixed bits is **0x41E2**.

Using this information we can now develop a software function that will calculate the appropriate attributes to be passed into the supplied Xil_SetTlbAttributes() function in the Xilinx BSP. An example of such a function is show below, which we will place in a new C source file called "mmu_control.c":

```
int adjust_mmu_mode(unsigned int start_of_1MB_address_region, unsigned int features)
{
	unsigned int mmu_attributes = 0;

	/* Declare the part of the page table value that gets written to the */
	/* MMU Table, which is always fixed. */
	/* NS = b0, Bit 18 = b0, TEX(2) = b1, Bit 9 = b0, Domain = b1111, */
	/* Bits(1:0) = b10 ... Equivalent hex value = 0x41e2 */
	const unsigned int fixed_values = 0x41e2;

	// Calculate the value that will be written to the MMU Page Table
	mmu_attributes = fixed_values + features;

	// Write the value to the TLB
	Xil_SetTlbAttributes(start_of_1MB_address_region, mmu_attributes);

	return (0);
}
```

Calling this function from a user's software application is now an extremely simple task, and does not require the software developer to remember any of the complicated bit positions related to the MMU page table. For example, if the developer wants to set the region of memory starting at address 0x20000000 to be non-global, non-cacheable, and have access permissions in "privileged mode" only, then they simple pass those macros into the function:

```
adjust_mmu_mode(0x20000000, NON_GLOBAL+NON_CACHEABLE+AP_PRIVIEGED_ACCESS_ONLY);
```

The chances of making a human error with this approach is drastically reduced, and the software developer wastes no time by performing the calculations manually. In the event of a software update in later years, this code is also considerably easier to read and understand.

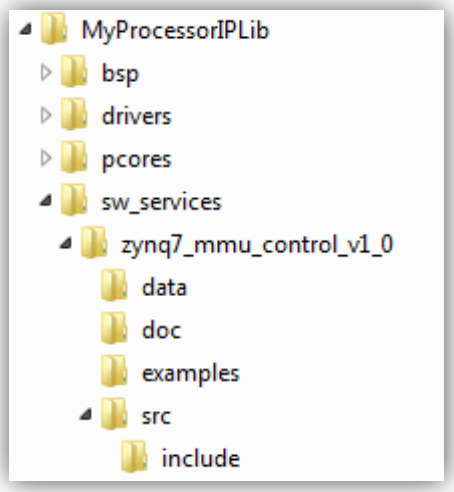
A prototype for this function is created in a header file (mmu_control.h), and this will be used later.

## Packaging the Code Into a Library

From a software coding perspective, we now have a working solution which could be manually added to future software projects. However, there are source files which must be copied to the correct directories, and each project would need to be manually updated if updates or bug-fixes were made. It would be a much nicer solution if we could add our MMU control functionality to other projects in the form of a library, allowing future users to access the software functions without copying raw C code from project to project.

Fortunately the Xilinx SDK framework allows us to achieve this task quite simply. To package code into a library requires that we create a known directory structure and also a few control files. We will explore these requirements in detail.

### The Directory Structure




The first requirement of a Xilinx software library is to place the source files into a known and pre-defined directory structure on the user's workstation. Readers with experience of creating custom processor peripherals and drivers may already be familiar with the "MyProcessorIPLib" folder containing "pcores" and "drivers" sub-directories. Custom libraries are placed in the same directory structure, but under a sub-directory called "sw_services". Inside the sw_services directory we can create any number of directories, each of which represent a unique software library. In the case of the example shown, we have created a library called "zynq7_mmu_control_v1_0"; note that we have optionally included a version number using the "v1_0" notation. Beneath each library folder, two additional

folders must be added; "data" which contains some important control files, and "src" which contains the source files for the library. The "src" directory can optionally contain sub-directories, and in our example we have placed our header file in the "src/include" directory. Two further directories can optionally be created in each library; the "doc" directory which can contain documentation for the library, and the "example" directory which can contain example software applications demonstrating use of the library.

## The MLD file

The next requirement for a custom software library is a control file called the Microprocessor Library Definition file (MLD). This definition file must be placed in the "data" directory, and must have the suffix <library_name>.mld . In our example we have created a file called "zynq7_mmu_control.mld".

```
OPTION psf_version = 2.1;

BEGIN LIBRARY zynq7_mmu_control
  OPTION drc = zynq7_mmu_control_drc;
  OPTION copyfiles = all;
  OPTION REQUIRES_OS = (standalone);
  OPTION SUPPORTED_PERIPHERALS = (cortexa9);
  OPTION APP_LINKER_FLAGS = "-Wl,--start-group,-lzynq7_mmu_control,-
lxil,-lgcc,-lc,--end-group";
  OPTION desc = "Zynq 7 MMU Control";
  OPTION VERSION = 1.0;
  OPTION NAME = zynq7_mmu_control;
END LIBRARY
```

The contents of the file are fairly simple, and start with a line that describes which version of the file syntax is being used (known as the "Platform Specification Format"). At the time of writing, the latest version is 2.1 and this is denoted using the syntax "OPTION psf_version = 2.1". The remainder of the text in the MLD is enclosed within the "BEGIN LIBRARY" and "END LIBRARY" lines, the first of which also features the name of the library and matches the file name and directory name, "zynq7_mmu_control". The majority of the MLD file syntax rarely changes from the default values, so the file contents shown above can be used as a starting point for custom libraries. The syntax is simply a list of parameters which are prefixed by the keyword "OPTION", and the various parameters are explained in the table below.

| Option | Description |
|---|---|
| DRC | Design Rule Check. The name of a procedure in an associated TCL file which performs DRC functionality. This is discussed later. |
| COPYFILES | Tells the Xilinx SDK library generation tools which of the source files should be copied into the source file tree of the board support package (BSP). In most cases this parameter can be set to "all", to copy all source files into the BSP. |
| REQUIRES_OS | Tells the Xilinx tools that this library is dependent on another BSP. In our example, we are adding additional functionality to the "standalone" BSP (also known as "bare metal" BSP). |
| SUPPORTED_PERIPHERALS | Some libraries might only be applicable to a specific piece of hardware in the embedded design. In our example we are writing functions which can only work with the Cortex A9 processor, so the library will only be compiled when that hardware is included in the design. If a MicroBlaze soft processor design were to be created, this library would not be available for the user to select. Another example of dependencies would be a library targeting Ethernet interfaces, and therefore should not be compiled into the BSP unless an Ethernet MAC was present in the design. |
| APP_LINKER_FLAGS | When software applications are compiled, the user is usually responsible for adding the "-l" switch to the compiler command line. This switch tells the compiler to include pre-compiled functions from a named library. This parameter allows the author of a library to have the –l switch added automatically to the compiler command line for any software applications that are created. In the case of our example we must add "-lzynq7_mmu_control", the name of our library. |
| DESC | Library description. This is a free text field containing a string that will be displayed in the BSP Settings of the Xilinx SDK GUI. |
| VERSION | The version number of the library. If multiple libraries exist in the directory structure with the same name but different version numbers, the Xilinx SDK GUI will present a way for the user to choose which version of the library they wish to use in their BSP. |
| NAME | The name of the library. This matches the name of the library directory and the prefix of the MLD filename; "zynq7_mmu_control". |

MLD files can also optionally contain user parameters which are denoted by the keyword "PARAM". The library in our MMU control example contains no parameters, but an example of the syntax is shown below for the purposes of completeness.

```
PARAM name = extra, desc = "Add extra functions?", type = bool, default = false ;
```

The purpose of a user parameter is to introduce flexibility into the library. In the very simple example shown above, a Boolean switch is implemented in a user parameter to allow the end user to control whether advanced functions should be compiled into the library.

From the end user's perspective, parameters are displayed in the Xilinx SDK GUI using additional menus which appear in the Board Support Package settings. User parameters can be useful in situations where, for example, it might be desirable to keep the compiled size of the library small unless the user required certain functions. User parameters are analysed and acted upon by the associated TCL script which, in the case of this example, could selectively decide which source files to compile into the library. User parameters are not limited to Boolean true/false data types; parameters could also capture other data types from the Xilinx SDK GUI such as integer values, or drop down box choices.

## The TCL file

At the heart of a custom software library is a TCL script which executes when the library is included for compilation in the BSP settings. The TCL script allows advanced functionality to be implemented as part of the build process, and is limited only by the imagination of the library creator. Source files can be selectively copied, moved, or even updated by the TCL script to include or remove functionality, parameters can be adjusted in header files, and messages can be printed to the console. If the author of the library has included user parameters as part of their MLD file, these parameters can be used to make decisions within the TCL script.

```
proc zynq7_mmu_control_drc {libhandle} {
	puts "Running DRC for Zynq7_MMU_Control library... \n"
}

proc generate {libhandle} {
}

proc post_generate {libhandle} {
	xgen_opts_file $libhandle
}

proc execs_generate {libhandle} {
}

proc xgen_opts_file {libhandle} {
	# Copy the include files to the include directory
	set srcdir [file join src include]
	set dstdir [file join .. .. include]

	# Create dstdir if it does not exist
	if { ! [file exists $dstdir] } {
		file mkdir $dstdir
	}

	# Get list of files in the srcdir
	set sources [glob -join $srcdir *.h]

	# Copy each of the files in the list to dstdir
	foreach source $sources {
		file copy -force $source $dstdir
	}
}
```

In this example we can see that the first procedure in the TCL file is a design rule check (DRC) called "zynq7_mmu_control_drc". This procedure name matches the

line in the associated MLD file, denoted by the "DRC" parameter. In this example the procedure does nothing more than print a line of text to the console in the Xilinx SDK, but the syntax of the procedure could implement any number of more advanced features if required.

Other procedures in the TCL file have reserved names, and these procedures are required by the Xilinx SDK tools. The "generate" procedure is called after library files are copied from their source folder into the BSP, the "post_generate" procedure is called after the "generate" has been called on all operating systems / drivers / libraries in the design, and the "execs_generate" procedure is called after all of the BSPs / libraries / drivers in the design have been generated. This system of pre-defined procedures allows the authors of libraries to implement a wide range of functionality at every stage of the software compilation process, even if the user library depends upon sources created and built outside of the custom library. In the case of this example, the only procedure to be populated with any code is the "post_generate" procedure, which calls an additional user-written procedure called "xgen_opts_file".

To re-cap; our "src" directory contains a single C source file (mmu_control.c) which contains the "adjust_mmu_mode" software function that we wrote earlier. We also created the mmu_control.h header file which contains all of the macros (#define statements) and the prototype for our custom function, and this was placed in the "src/include" directory. The "xgen_opts_file" procedure checks for the existence of a destination directory, creating it if necessary, and then copies each of the source files into the correct destination directory within the BSP.

## The Makefile

The last control file is the Makefile, which is located in the "src" directory. Makefiles are common in software compilation flows, running the compiler tools to build output files based on dependencies. It is beyond the scope of this document to fully document the operation of a Makefile; extensive documentation is available at https://www.gnu.org/software/make/manual/make.html but we shall discuss the basics as an aid to understand the required changes to this file.

Makefiles operate on a principle of "rules", and each rule has "targets" and "dependencies". Each rule has a header line which is formed of one or more keywords separated by a colon ":" character. The Makefile syntax uses a target before the ":" character, and a dependency after the ":". Immediately following this line is the "recipe" that creates the target. Another way to consider a "dependency" is to consider it as a "prerequisite" that must already be in place before the rule can execute.

```
target ... : dependency ...
             recipe starts here
             commands and operations
             ...
```

```
COMPILER=
ARCHIVER=
CP=cp
COMPILER_FLAGS=
EXTRA_COMPILER_FLAGS=

RELEASEDIR=../../../lib
INCLUDEDIR=../../../include
INCLUDES=-I${INCLUDEDIR}
ZYNQ7_MMU_CONTROL_DIR = .

LIB_SRCS = $(ZYNQ7_MMU_CONTROL_DIR)/mmu_control.c

ZYNQ7_MMU_CONTROL_SRCS = $(LIB_SRCS)

ZYNQ7_MMU_CONTROL_OBJS = $(ZYNQ7_MMU_CONTROL_SRCS:%.c=%.o)


EXPORT_INCLUDE_FILES = $(ZYNQ7_MMU_CONTROL_DIR)/include/mmu_control.h

libs: libzynq7_mmu_control.a
   cp libzynq7_mmu_control.a $(RELEASEDIR)
   make clean

include:
   @for i in $(EXPORT_INCLUDE_FILES); do \
   echo ${CP} -r $$i ${INCLUDEDIR}; \
   ${CP} -r $$i ${INCLUDEDIR}; \
   done

clean:
   rm -rf obj/*.o
   rmdir obj
   rm libzynq7_mmu_control.a

libzynq7_mmu_control.a: obj_dir print_msg_isf_base $(ZYNQ7_MMU_CONTROL_OBJS)
   @echo "Creating archive $@"
   $(ARCHIVER) rc $@ obj/*.o

obj_dir:
   mkdir obj

print_msg_isf_base:
   @echo "Compiling Zynq7_MMU_Control Library"

.c.o:
   $(COMPILER) $(COMPILER_FLAGS) $(EXTRA_COMPILER_FLAGS) $(INCLUDES) -c $< -o obj/$(@F)
```

The contents of a Makefile can be executed from the command line simply by typing "make", and the tools will look for a default makefile called "makefile" (with no file extension) in the current working directory. The first rule that is found in the file (e.g. "libs") is the first to be executed. Alternatively rules can be executed individually from the command line using the command "make <target_name>". Let's begin by reviewing a simple rule.

```
libs: libzynq7_mmu_control.a
      cp libzynq7_mmu_control.a $(RELEASEDIR)
      make clean
```

Here we can see that we have a target called "libs". The target doesn't output a specific file per se, but the rule is dependent on the existence of a file called "libzynq7_mmu_control.a". The target is created / achieved by executing two lines in the recipe; the first of which is a simple copy command (cp) which copies the

"libzynq7_mmu_control.a" file to a directory denoted by the variable "RELEASEDIR". Variables can be referenced by the use of a dollar ($) character and then the name of the variable in round brackets. The second line of the recipe executes another Makefile rule called "clean". The variable "RELEASEDIR" was declared and assigned further up in the Makefile.

A considerably more complex rule is shown at the bottom of the Makefile. This is a more sophisticated rule and uses a variety of advanced syntax features.

```
.c.o:
	$(COMPILER) $(COMPILER_FLAGS) $(EXTRA_COMPILER_FLAGS) $(INCLUDES) -c $< -o obj/$(@F)
```

The target name denotes a special case which is known as a "suffix rule". In essence, this type of rule creates wildcard targets of object files (.o) from C source files (.c). So the target "example.o" would be created by the recipe using the source file of the same prefix name "example.c" and, similarly, "my_code.o" would be created from "my_code.c". The recipe executes the command $(COMPILER) which in our case would be the GCC compiler, using compiler flags listed in the variables $(COMPILER_FLAGS) and $(EXTRA_COMPILER_FLAGS), specifying the include file switch (-I) from the variable $(INCLUDES). The "-c" switch instructs the compiler to compile the sources into object files but not link them into an executable, and the "-o" switch tells the compiler to output the object file(s) into a directory called "obj" and into filenames denoted by "$(@F)". The "$/<" syntax denotes the name of the first dependency, which in the case of our suffix rule example is the name of the .c source file that is to be compiled. In the interests of completeness, "$(@F)" is another special item of Makefile syntax which is called an "Automatic Variable". Specifically, the "@F" denotes the name of the file in the target name, whereas "$@" would be the name of the target itself. This Makefile is sadly even more confusing because variables $(COMPILER), $(ARCHIVER), $(COMPILER_FLAGS and $(EXTRA_COMPILER_FLAGS) are apparently all declared but have no values. This is because their values get inherited from another Makefile that is built into the Xilinx SDK tools, and which calls our user Makefile. This is done to make the compilation of our library flexible enough to be compatible with different compilers (ARM Cortex A9, PowerPC 405, Microblaze, etc).

In summary, the use of these Makefile features allows any number of C source files to be added to the "src" directory, and they would all automatically be compiled into object files (.o) using the correct compiler settings which are provided by the Xilinx SDK. To edit this Makefile for our custom library, we simply had to update the makefile in a few places:

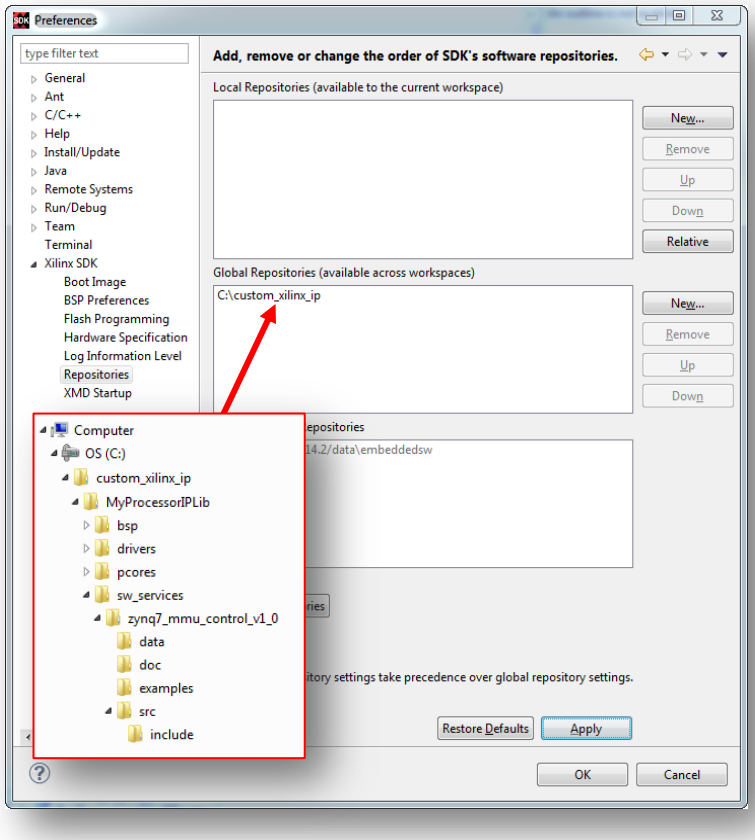
- Edit the "LIB_SRCS" variable to list the source files in our library. Separated by spaces, as necessary.
- Update the filename of the archived library, in our case "libzynq7_mmu_control.a".

- Update some of the variable names to reflect the name of our custom library, e.g. ZYNQ7_MMU_CONTROL_DIR, ZYNQ7_MMU_CONTROL_SRCS, and ZYNQ7_MMU_CONTROL_OBJS.
- List the names of our custom header files in the EXPORT_INCLUDE_FILES variable. (e.g. "mmu_control.h")
- Change the string that is printed to the console during compilation (e.g. "Compiling Zynq7_MMU_Control Library").

## Testing and Using the Custom Library

Before any custom library can be used in the Xilinx SDK, it is important to configure the SDK so that it can locate custom libraries. Many readers of this guide will be familiar with configuring the SDK to look for user repositories containing custom drivers, and this is precisely the same flow.

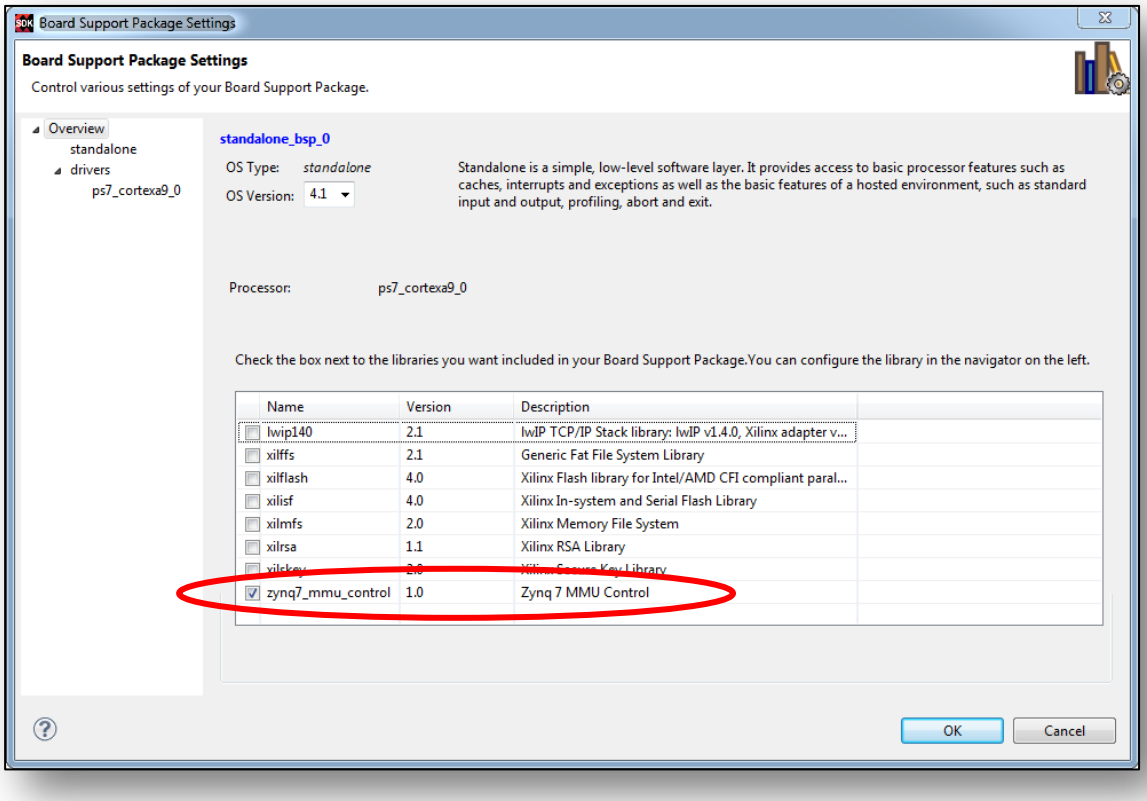
In the SDK, open the "Preferences" dialogue by choosing "Window → Preferences" from the menu bar. Select the "Xilinx SDK → Repositories" pane, as shown in the screenshot.



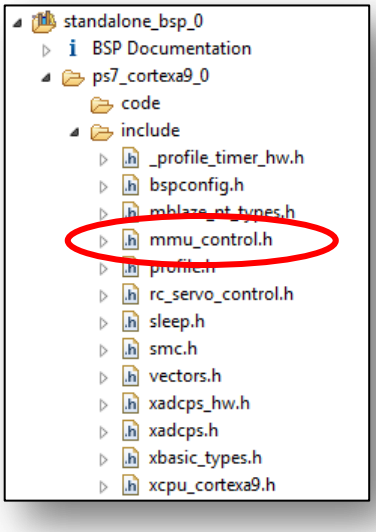

Add a new repository to the list by clicking the "New..." button next to the right of the "Global Repositories" list, and point to the folder that you created for your custom library. The folder you must choose here should be the level of the file system above the "MyProcessorIPLib" folder. The folder structure is critical when defining the location of a global repository, and the directory names "MyProcessorIPLib" and "sw_services" are reserved and required. Click the "Rescan Repositories" button, followed by "OK". This setting will provide visibility of your custom library to the SDK tool, and will enable your library to be selected in the BSP settings. There is an addition list shown on this screen called "Local Repositories"; this performs an identical function to adding the library to the "Global Repositories" list, with the exception that the library will only be visible to the current SDK workspace rather than having global scope across all workspaces. The purpose of a custom library is

to allow the same software functions to be used across many projects, so the use of a global repository is more useful than a local one in most cases where custom libraries are created.

The setup for the library is complete and we can now check to make sure that the custom library appears in the GUI for the Standalone BSP Settings. In the Xilinx SDK, either create a new BSP or right-click an existing one and choose "Board Support Package Settings". In the "Overview" tab of the BSP settings you will now be able to select your new library to include it in the BSP. Just tick the box.




The BSP will automatically re-compile and in the "include" directory you will see that the custom header file (mmu_control.h) has been added to the list of other BSP header files.




That's it! The custom library can now be used in any software application projects by "including" it, just like any other header file from the BSP.

```
#include "mmu_control.h"
```

A set of resource files is provided with this document as a reference, including the custom library created in the example described.