cādence

Cadence Driver Porting Guide

Product Version 1.5

April 2020

Cadence Driver Porting Guide

Cadence Design Systems

Table of Contents

List of Figures

Chapter 1. Acronyms

API	Application Programming Interface.
Core Driver	Cadence Firmware component that provides IP programming abstraction.
CPS	Cadence Platform Services. A set of basic platform-specific access functions acting as hardware abstraction layer for the core driver to operate.

Chapter 2. Overview

2.1. Document Purpose

The purpose of this document is to outline how to perform the following porting tasks related to Cadence drivers:

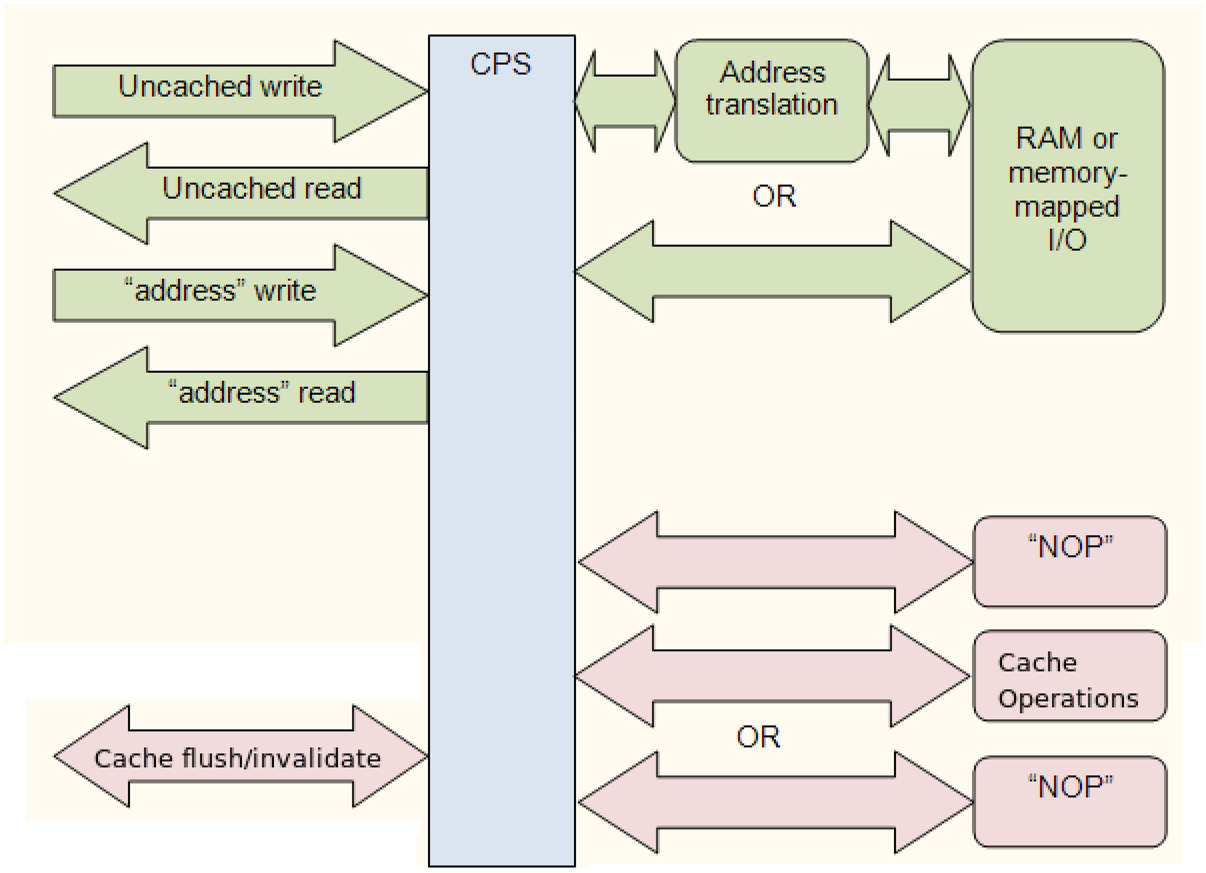
- Implementing CPS for your platform. This need be done only once to enable all Cadence drivers for your platform.
- Implementing the Common Driver Interface for your platform. This is the platform-specific code to connect a given Cadence driver to the rest of your system. Depending on your platform, aspects of this will be specific to the platform or the layer above the driver.

2.2. CPS Overview

Cadence Core Drivers are built on a common hardware abstraction layer, called Cadence Platform Services (CPS). This is where all aspects specific to a CPU are handled. Specifically, the following is handled by CPS:

- Uncached access (i.e. writing through the cache with order preserved) to memory, used for hardware registers and DMA descriptors.
- Translating addresses (if needed) between the CPU's view (when running the driver), and hardware's view. This is used for things such as DMA descriptors, where a buffer address as seen by hardware may be different than that seen by the CPU (e.g. if an MMU is being used).
- Cache flush/invalidate operations (if needed), for example if the hardware needs to access memory directly for DMA operations.

Figure 2.1. Pictorial Representation of CPS



Each platform requires an implementation of CPS. Once CPS is implemented for a platform, all Cadence Core Drivers can use it on that platform. This reduces porting effort in case you acquire further drivers from Cadence.

2.3. Common Driver Interface Overview

Several important aspects of using a driver depend heavily on the platform:

- How the RAM used by the driver must be allocated. On a simple system, it may be sufficient to reserve a block of RAM statically, while on a full memory-protected OS, it may be necessary to use a kernel allocation function.
- How the memory-mapped I/O range for the hardware's registers is obtained. On a simple system, it may be hard-wired, while on other systems it may be necessary to call a mapping function. If the hardware is on a plug-and-play bus such as PCI, the bus enumerator may assign this, requiring a fundamentally different way of connecting the driver.
- How the interrupt number/vector for the hardware is obtained. On a simple system, it may be hard-wired, while on other systems it may be necessary to call a mapping function. If the hardware is on a plug-and-play bus such as PCI, the bus enumerator may assign this, requiring a fundamentally different way of connecting the driver.

Due to the above considerations, for maximum portability:

- Cadence drivers never allocate memory - this must be done by your integration code and passed to the driver.

- Cadence drivers never assume a location for the underlying hardware's registers - this must be determined by your integration code and passed to the driver.
- Cadence drivers never assume a particular interrupt "topology" - your integration code must arrange for the driver's ISR to be called upon receipt of the underlying hardware's interrupt.

This is the Common Driver Interface.

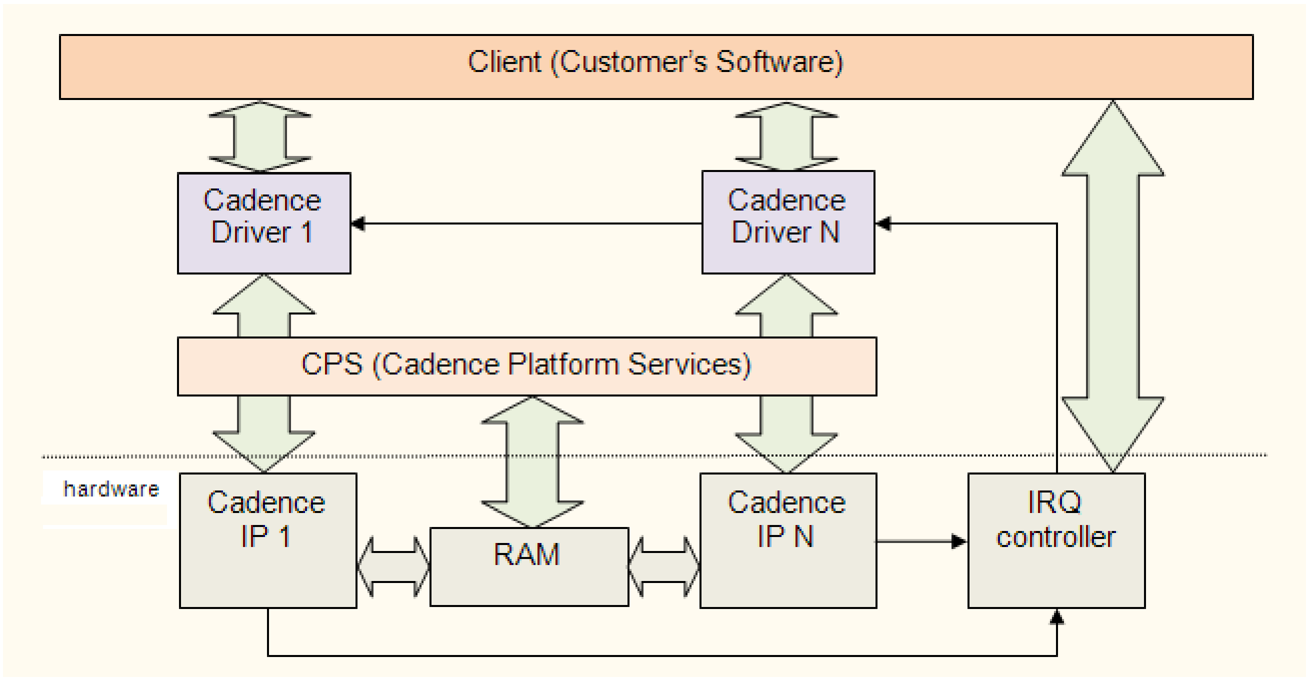
2.4. Use Model

Choosing any aspect of a computing platform, one can find a spectrum of possible implementations. It is simply not possible to list every situation. Instead, the following three classifications of systems are intended to serve as a rough guideline for the tasks required to enable a Cadence driver for your platform.

2.4.1. Bare-Metal Architecture

The following depicts the typical scenario for a simple "bare-metal" platform (a platform not using an MMU, without a formalized porting layer or technology stacks):

Figure 2.2. Bare-Metal Architecture



In such a system, it is likely that your custom software will use the driver API directly, and you may have direct control over the interrupt controller setup. Since there is no MMU it is likely that there will be no data cache available. Implementing CPS on such a platform will typically involve:

- Using simple constructs such as

  ```
  *(volatile uint32_t*)address = value
  ```

  for the memory-mapped I/O functions
- Using the same for the address-translation functions

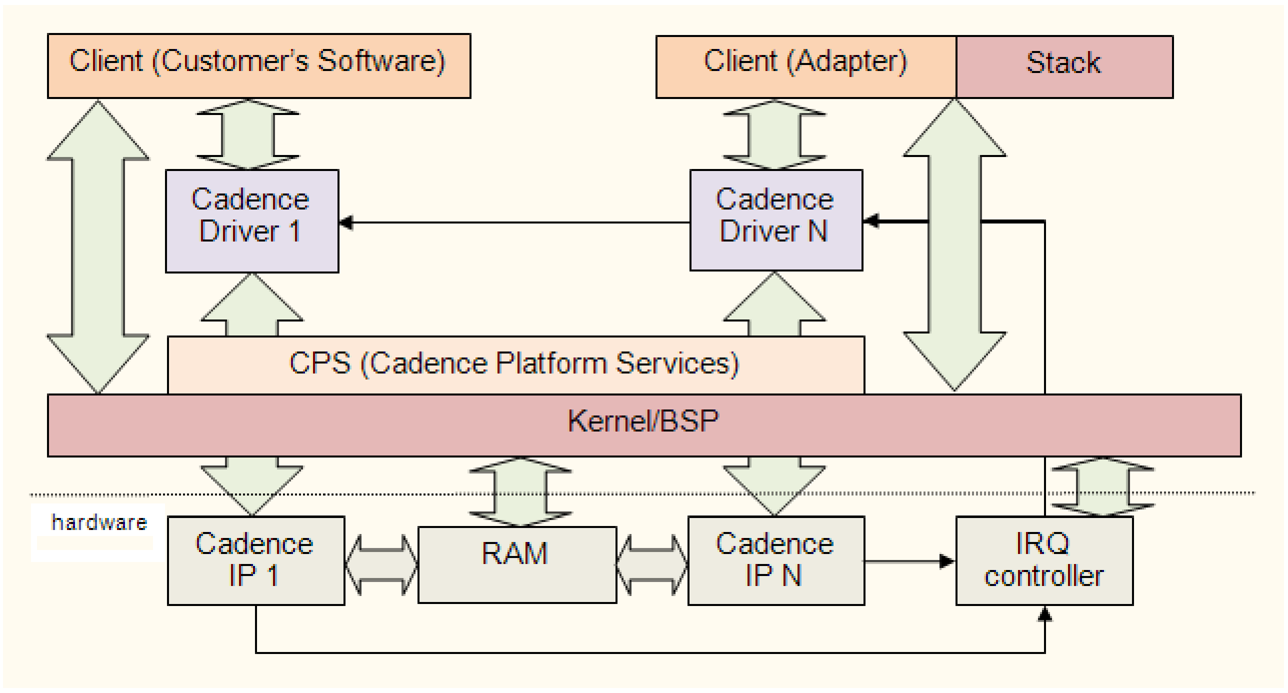
- Implementing do-nothing functions for cache flush and invalidate

Implementing the integration code on such a platform will generally be straightforward unless the hardware is on a plug-and-play bus.

2.4.2. Microkernel Architecture

If you are using an RTOS or microkernel, there may or may not be a standard stack for a given technology. If there is a standard stack, it is likely you would want to create a small adapter layer to convert between the Cadence driver's API and the lowest layer of that stack. We will again assume no MMU is being used on such a platform. Also on such a platform, it is likely that there is some level of support from the kernel or BSP for programming the interrupt controller attaching handlers, and possibly memory-mapped I/O services exist.

Figure 2.3. Microkernel Architecture



Implementing CPS on such a platform will typically involve:

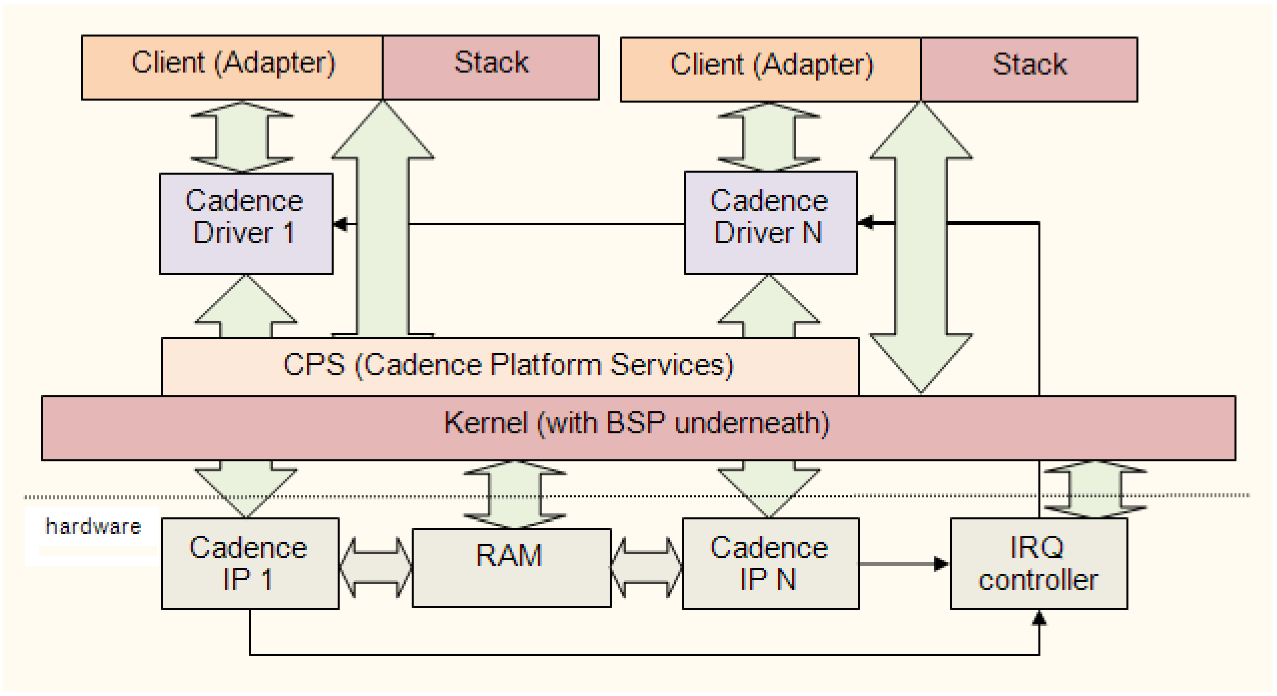
- For memory-mapped I/O, either using simple constructs as in the bare-metal example, or calling service functions in the kernel or BSP
- Using the same for the address-translation functions
- Implementing do-nothing functions for cache flush and invalidate

Implementing the integration code on such a platform will generally be straightforward unless the hardware is on a plug-and-play bus.

2.4.3. Full OS Architecture

If you are using a full memory-protected OS (definitely an MMU being used), there will typically be a standard stack for a given technology. Therefore, you will likely want to create a small adapter layer to convert between the Cadence driver's API and the lowest layer of that stack. Also on such a platform, it is typical that interrupt attachment and memory-mapped I/O access will be abstracted for you.

Figure 2.4. Full OS Architecture



Implementing CPS on such a platform will typically involve:

- For memory-mapped I/O, calling service functions in the kernel. This usually involves mapping the hardware's addressable region at driver initialization and then using it, either directly or through I/O services, in other driver functions.
- For the address-translation functions, calling translation functions in the kernel
- Implementing cache flush/invalidate functions using the OS if the hardware accesses memory directly, e.g. for DMA.

On this type of platform, it will be necessary to wrap the Cadence driver to create a driver package which follows the rules of the OS driver framework. This usually means:

- Creating an initialization function conforming to the framework's rules, in which you acquire resources and call the Cadence driver's init function. All Cadence drivers provide probe and init functions to aid in resource acquisition.
- Creating other functions to start, stop and destroy the driver, again calling the corresponding functions in the Cadence driver.
- Calling other Cadence driver functions as required, either from driver framework functions or stack functions.

2.5. Concurrency Models

Cadence Core Drivers are architected such that they are portable to platforms which enforce a particular concurrency model, while also giving you the maximum freedom to choose the model suitable for your application. Also, the minimal assumptions made allow for maximum efficiency on simple platforms. Part of the API for most drivers involves calling back your code. The basic rule is that these callbacks are made from the driver's interrupt handler. This is what yields the freedom and portability just mentioned.

Chapter 3. Common Driver Interface

3.1. Overview

Each Cadence Core Driver represents a different technology and accordingly, offers varying functionality. However, all Cadence drivers share a common framework of functions, to quickly get you beyond the basic bringup and on to making use of the Cadence technology you have acquired.

The following functions are common to all Cadence drivers:

- `probe(configurationInfo, resourceRequirements)` - this is the first function you should call. Although the exact signature varies by driver, the pattern is that you pass information about how the driver should be used, and it fills information about the required resources (RAM) to support that configuration. In a dynamic system or where your code is to be ported to a variety of platforms, you may wish to call probe repeatedly to "negotiate" an acceptable configuration that fits within platform resources. The configuration includes the base address of the hardware's registers, so the driver may read identification from the hardware during this call (to determine resource requirements).

- `init(instanceData, configurationInfo)` - after a successful probe(), call this function, passing a pointer to an allocated block of RAM of the required size (the required size is filled by probe), and the configuration again (so the driver may copy anything it must remember).

- `start(instanceData)` - after a successful init, you should attach the driver's interrupt handler (isr) and enable the interrupt at the interrupt controller. Then, call this function to start the driver. It will enable relevant interrupts at the source.

- `isr(instanceData)` - this is the driver's interrupt handler, which your code must attach before calling start()

- `stop(instanceData)` - this will disable any hardware interrupts at the source

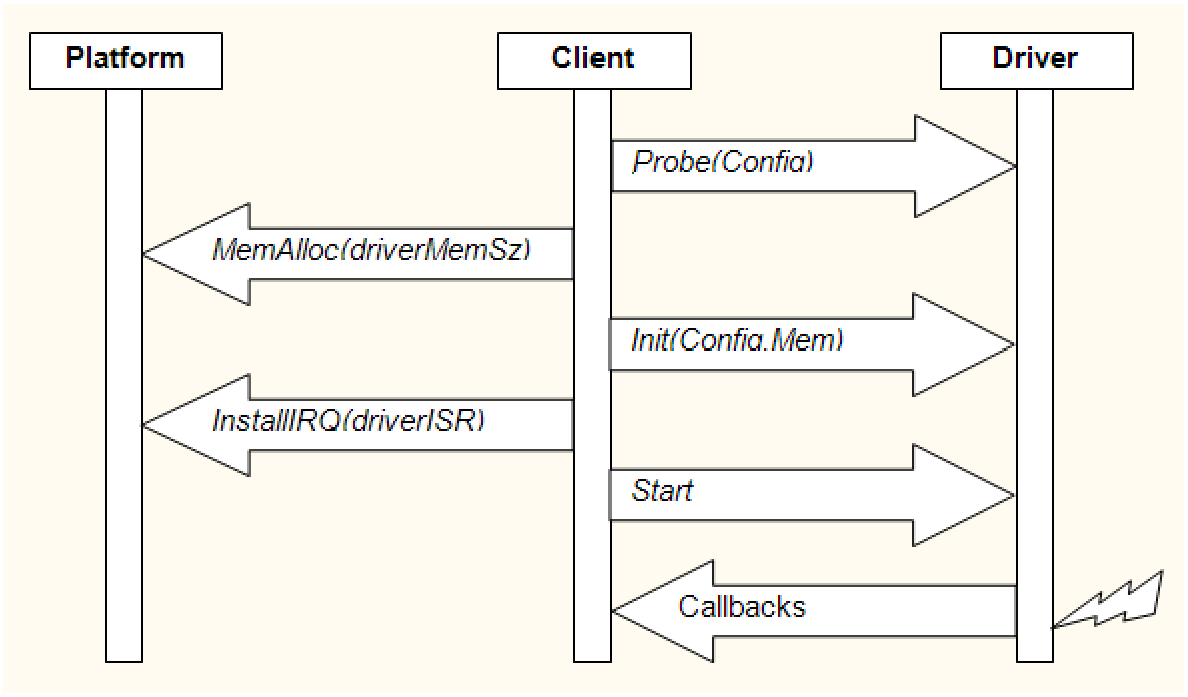
- `destroy(instanceData)` - this will internally stop, and put the driver in a non-responsive state until init is called again

The instanceData parameter is a pointer to a RAM block, allocated by the client before calling `init()`, based on the resource requirements filled by `probe()`. The reason for this mechanism is to allow for multiple instances of the driver, in case you have more than one of the same type of Cadence controller in your system.

3.2. Driver Initialization & Startup

The common aspects of driver initialization and startup are illustrated below:

Figure 3.1. Driver Initialization & Startup Sequence



3.3. Concurrency Model Details

Cadence drivers do not support lock/unlock mechanism. If you are working on multithreaded or multicore environment you should implement these functions for your system and wrap all calls to the driver.

Chapter 4. Sample CPS implementation for Bare Metal

4.1. Overview

The code here implements the Cadence Porting System for a bare-metal environment. This should be applicable to most bare-metal systems where there is no MMU or data cache.

4.2. Sample code

```
uint32_t CPS_ReadReg32(volatile uint32_t* address) {
    return *address;
}

void CPS_WriteReg32(volatile uint32_t* address, uint32_t value) {
    *address = value;
}

uint8_t CPS_UncachedRead8(volatile uint8_t* address) {
    return *address;
}

uint16_t CPS_UncachedRead16(volatile uint16_t* address) {
    return *address;
}

uint32_t CPS_UncachedRead32(volatile uint32_t* address) {
    return *address;
}

void CPS_UncachedWrite8(volatile uint8_t* address, uint8_t value) {
    *address = value;
}

void CPS_UncachedWrite16(volatile uint16_t* address, uint16_t value) {
    *address = value;
}

void CPS_UncachedWrite32(volatile uint32_t* address, uint32_t value) {
    *address = value;
}

void CPS_BufferCopy(volatile uint8_t *dst, volatile const uint8_t *src, uint32_t size) {
    memcpy((void*)dst, (void*)src, size);
}

void CPS_CacheInvalidate(void* address, size_t size, uintptr_t devInfo) {
    return;
}

void CPS_CacheFlush(void* address, size_t size, uintptr_t devInfo) {
    return;
}
```

Chapter 5. Sample CPS implementation for Linux

5.1. Overview

The code here implements the Cadence Porting System for a Linux environment. This should be applicable to Linux kernels from 2.6.32 onwards with minor modifications for your system. Note that locks are not implemented in this sample.

If a Cadence Driver makes use of the CacheFlush() and CacheInvalidate() functions the driver documentation will provide details of how to pass the "device" pointer to the Core Driver so that it will be available to your implementation of CPS.

5.2. Sample code

```
uint32_t CPS_ReadReg32(volatile uint32_t* address) {
    return readl(address);
}

void CPS_WriteReg32(volatile uint32_t* address, uint32_t value) {
    writel(value, address);
}

uint8_t CPS_UncachedRead8(volatile uint8_t* address) {
    return readb(address);
}

uint16_t CPS_UncachedRead16(volatile uint16_t* address) {
    return readw(address);
}

uint32_t CPS_UncachedRead32(volatile uint32_t* address) {
    return readl(address);
}

void CPS_UncachedWrite8(volatile uint8_t* address, uint8_t value) {
    writeb (value, address);
}

void CPS_UncachedWrite16(volatile uint16_t* address, uint16_t value) {
    writew (value, address);
}

void CPS_UncachedWrite32(volatile uint32_t* address, uint32_t value) {
    writel (value, address);
}

void CPS_BufferCopy(volatile uint8_t *dst, volatile const uint8_t *src, uint32_t size) {
    memcpy((void*)dst, (void*)src, size);
}

/**
 * Invalidate cache using Linux DMA sync routines
 * @param[in] address Virtual address of buffer to sync. Buffer must
 *    be allocated using dma_map_single() function
 * @param[in] size Size of buffer
 * @param[in] device Pointer to struct device* which owns buffer
 */
void CPS_CacheInvalidate(void* address, size_t size, uintptr_t device) {
    uintptr_t phys_address = __pa(address);
    struct device *dev = (struct device *) device;
    dma_sync_single_for_cpu(dev, phys_address, size, DMA_BIDIRECTIONAL);
}

/**
 * Flush cache using Linux DMA sync routines
 * @param[in] address Virtual address of buffer to sync. Buffer must
 *    be allocated using dma_map_single() function
 * @param[in] size Size of buffer
 * @param[in] device Pointer to struct device* which owns buffer
 */
void CPS_CacheFlush(void* address, size_t size, uintptr_t device) {
    uintptr_t phys_address = __pa(address);
    struct device *dev = (struct device *) device;
    dma_sync_single_for_device(dev, phys_address, size, DMA_BIDIRECTIONAL);
}
```