SafeXcel IP™
Driver Framework v4.2

Porting Guide

Document Revision: B
Document Date: 2013-02-14
Document Number: 007-900420-304
Document Status: Accepted

INSIDE Secure B.V.
Boxtelseweg 26A
5261 NE Vught
The Netherlands
Phone: +31-73-6581900
Fax: +31-73-6581999
http://www.insidesecure.com/

For further information contact: ESSEmbeddedHW-Support@insidesecure.com

Revision History

Doc Rev	Page(s) Section(s)	Date	Author	Purpose of Revision
A	All	2011-09-02	RWI MHO	• Created document from Driver Framework v4.1 Porting Guide Rev B. • Editorial changes and template update. • Updated the API-states figures. • Added test suite description (chapter 2.5). • Updated text in the introduction chapters (1 and 2).
B	All	2013-02-14	FvdM	• Update template

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

1.1 Target Audience

This document is intended for software developers who need to implement or port the Driver Framework environment. Chapter 2 introduces the architecture and describes the components that surround the framework. Chapter 3 specifies the services required from the Driver Framework implementation.

1.2 Abbreviations and Definitions

AHB	Advanced High-performance Bus
API	Application Program Interface
blocking function	An API function is blocking when it returns first when the result is available.
bounce buffer	An area of low memory for a device to write and read data, from where the data can be copied to a higher memory area.
concurrent	Occurring within the same time frame / overlapping.
CPU	Central Processing Unit
DF	Driver Framework
DMA	Direct Memory Access
dynamic resource	A memory buffer allocated run-time by the host CPU in the system memory external to the EIP device that the latter can use for DMA operations.
EIP	Embedded Intellectual Property
EIP device	A processing entity implemented in hardware, also called EIP hardware block in this document for which a driver needs to be developed. In fact this can be any I/O hardware block interfaced with the host and the terms • EIP hardware block • EIP device • I/O hardware block can be used interchangeably in this document.
execution context	The context maintained by OS and/or host CPU for a program (like driver code) being executed by the host CPU.
host	The environment (HW and OS) that executes the driver software (device control functionality) and uses the services (hosts) of one or more EIP devices.
HW	Hardware
ISR	Interrupt Service Routine
LRU	Least Recently Used
memory domain	Address space within a computing environment where all resources used by the application are uniquely identified by their addresses. Same resources may have different addresses in different memory domains. Different resources may have the same addresses in different memory domains. Memory domain examples are Linux user process, Linux kernel domain.
OS	Operating System
PCI	Processor Component Interface
PLB	Processor Local Bus
re-entrant	An API function is re-entrant when it can safely be invoked from concurrent execution contexts simultaneously, e.g. the same code is executed in different execution contexts at the same time. Similar situation occurs when the API code being executed in one execution context is pre-empted by another execution context that also starts executing the same API code.
SW	Software

2 Functional Overview

This chapter provides a high-level overview of the software and hardware architecture and the position of the Driver Framework.

2.1 Architectural Overview

The hardware and software components for a typical driver environment are shown in Figure 1.

The EIP hardware blocks are standard components that are integrated into a hardware platform. Each EIP Driver Library is also a well-tested and standard component that is solely responsible for one of these EIP hardware blocks.

The Adapter deals with the OS integration and provides the driver API to the applications. In systems with concurrent applications, the Adapter must handle the context synchronization.

The Driver Framework provides the hardware integration for the driver environment, such as register access and DMA buffer management. The Driver Framework must implement the API specified in this document.

The Adapter and Driver Framework provide decoupling from integration issues and allow to EIP Driver Libraries to be ported to the driver environment *without* modification. This way, correct operation with the EIP hardware blocks is more likely and the porting effort is greatly reduced.

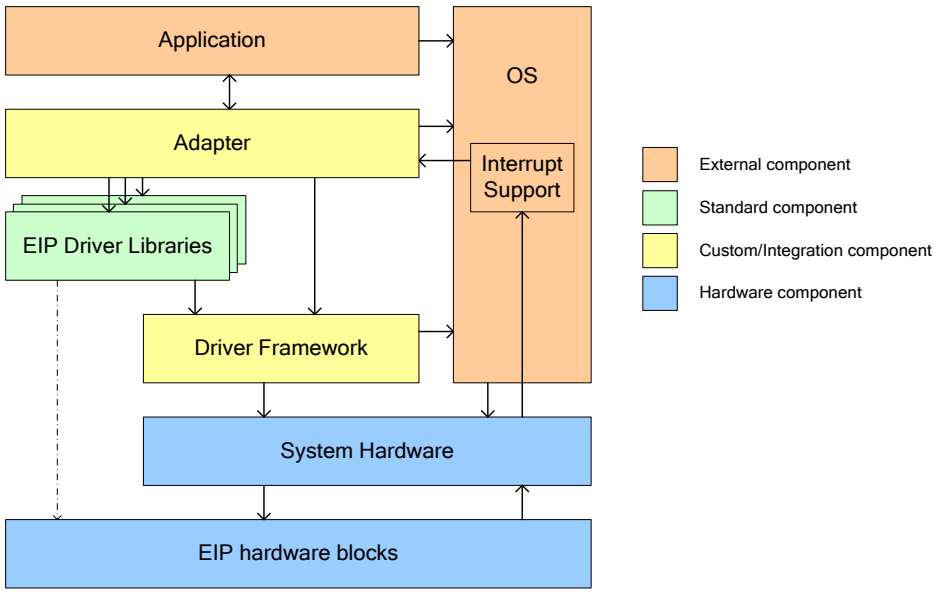

Figure 1 Architecture Overview

2.2 Adapter

The Adapter is a software layer between the application(s) and the EIP Driver Libraries, implementing whatever Application Program Interface (API) is defined for the applications. It adapts the driver's API to the EIP Driver Libraries APIs. All aspects of the driver's API are handled in the Adapter: blocking or asynchronous operation, operation queuing, related memory management, etc.

Since the EIP Driver Libraries are designed to be free of any integration issues, their respective APIs can put requirements on the Adapter to handle issues like resource sharing, concurrency and context synchronization. Interrupts from the EIP hardware blocks are closely related to context synchronization and are therefore installed and serviced by the Adapter. The Adapter can use the OS services to implement the interrupt handling and concurrent context synchronization.

The following tasks are typically handled in the Adapter layer:

- Application interfacing and multiple application instances support,
- Application operation queuing,
- Concurrent context synchronization,
- Cache coherency handling (see section 2.2.1),
- Dynamic memory allocation,
- Address translation for resources used in different memory domains,
- Endianness conversion between host CPU and EIP device for dynamic resources,
- Interrupt hooking and handling (see section 2.2.2).

Not all of these tasks may be required for the Adapter layer for a particular driver. The Adapter uses services from the Driver Framework (see section 2.4 and chapter 3) to handle some of these tasks. It should also be noted that the Adapter, unlike the EIP Driver Library, must be custom-made for every software (like OS) and hardware environment, together with the Driver Framework implementation.

2.2.1 Cache Coherency Handling

The `DMAResource_PreDMA()` and `DMAResource_PostDMA()` functions should be called as suggested by their names: before the EIP device accesses (by means of DMA) the shared buffer and after the EIP device has finished accessing the buffer. For some buffers these calls are made by the EIP Driver Libraries, while for other buffers the Adapter layer makes the calls at the appropriate time and in the appropriate execution context. This typically means outside the critical timing path, possibly as early as possible for the `DMAResource_PreDMA()` function and as late as possible for the `DMAResource_PostDMA()` function.

The implementation of these functions can address system aspects that are not known to the EIP Driver Library and the corresponding EIP device. For example, in Figure 2 below, the CPU data cache requires manual cache coherency in the path from the host to the system memory. Although slightly less common, the data cache can also be used on the path from the EIP device to the system memory.

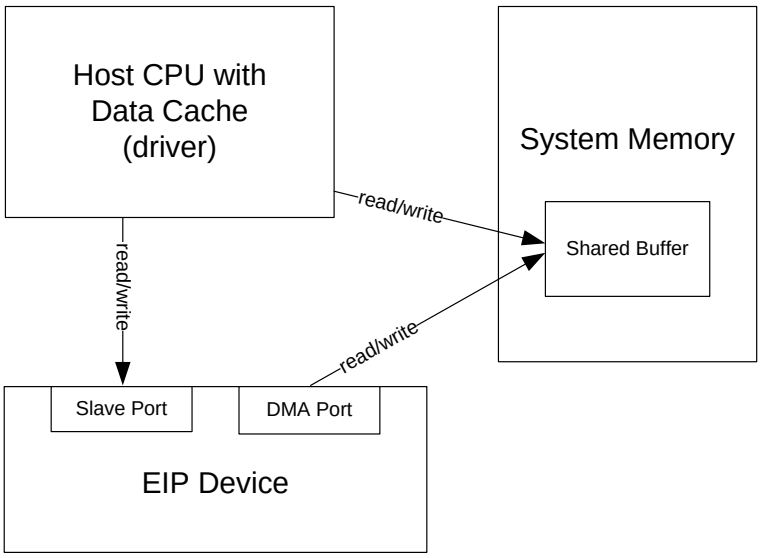

Figure 2 Data and control paths in a system

In the example shown in this picture the host CPU can write some data to the buffer in the system memory that is shared with the EIP device. Depending on the host CPU data cache architecture, this data may not be written immediately into the system memory but may get stored in the CPU's data cache instead. If the host CPU subsequently requests the EIP device to start the DMA read operation from the system memory for that shared buffer afterwards then the EIP device will read

obsolete data from the shared buffer. In this situation, it can be said that the host CPU and the EIP device have incoherent views on the data in the shared buffer in the system memory. To prevent this cache coherence issue the host CPU must commit the contents of its data cache to the system memory before requesting the DMA read operation on that shared buffer.

2.2.2 Interrupt Hooking and Handling

It is important to understand the dependency between the Adapter and the implementation of the Driver Framework in order to support interrupt contexts. Although the exact definition of interrupt context can be OS- and host CPU-specific, these contexts typically put significant constraints on the code:

- Interrupt handling must complete as soon as possible to keep interrupt latency low (this is especially important in a real-time embedded system), typical ISR execution time is a few microseconds;
- Interrupt handling cannot call functions yielding to OS scheduler in case the latter is unaware of such contexts (cannot schedule them).

To support execution in the interrupt context fully, the EIP Driver Libraries must fulfill the above-mentioned constraints. In cases where this is not possible, it is documented extensively. In addition, it is also the responsibility of the driver developer to ensure that these constraints are fulfilled during the implementation of the Driver Framework API.

Finally, some parts of the Adapter code can also execute in the interrupt context. This is completely under the control of the driver developer and does not affect the EIP Driver Libraries, except that certain preconditions related to resource sharing and API re-entrance must be fulfilled. This might require careful design of the Adapter, for example to avoid using OS functions yielding to OS scheduler in the interrupt context.

2.3 EIP Driver Libraries

One EIP Driver Library exists for each EIP hardware block. The EIP Driver Library is the only component allowed to access the registers and embedded memories of the corresponding EIP hardware block. In Figure 1, the dashed line from the EIP Driver Library to the EIP hardware shows that the EIP Driver Library is aware of the EIP hardware block internals. The EIP Driver Libraries provide an API that allows using the EIP hardware functionality without having to worry about the details of the EIP hardware.

The EIP Driver Library is a building block that is free from all the typical integration issues as mentioned in section 2.2 and abstracts the EIP hardware block, thereby removing the need for a driver developer to understand the EIP hardware block internals.

Each EIP Driver Library is designed according to the external storage and timeless principles. The external storage design principle means that the library does not make any assumptions about what memory is available and uses memory provided by the Adapter for internal administration. The timeless design principle means that the library does not contain any time-related loops and wait mechanisms.

2.4 Driver Framework

The Driver Framework provides decoupling from integration issues such as CPU specifics, compiler specifics and hardware platform details. This allows the EIP Driver Library to be a standard, reusable component. The Driver Framework is a set of APIs that can be used by the Adapter and the EIP Driver Libraries. These APIs can be classified as follows:

- **Basic Definitions API**: Types with guaranteed storage size and a few support macros;
- **C library API**: Compiler abstraction, guarantees the existence of a subset of the C library API;
- **Device API**: Provides access to static resources shared with EIP devices (see section 2.4.1, typically registers and memories implemented in the EIP device);

- **DMA Resource API**: Provides access to dynamic resources shared with EIP devices (see section 2.4.2, typically memory buffers available outside the EIP device, applicable only when the EIP device is DMA-capable).

The Driver Framework APIs must be implemented for different software and hardware environments. This document is the specification of these APIs and is a source of information for implementing or porting drivers based on the EIP Driver Libraries to new environments by means of implementing the Driver Framework API for those environments. The EIP Driver Libraries do not need to be changed during such porting as long as the corresponding EIP hardware blocks remain unchanged.

2.4.1 Device API and Static Resources

Every EIP device has a pre-defined number of static resources, typically the registers and memories implemented in the device. These resources are static and will not change at run time; their absolute addresses in the memory map have been defined during hardware integration. Another important property of static resources is that they do not require cache coherence handling in most systems, e.g. static resources are not cached either by the host CPU or by the EIP device.

The Device API is intended for handling these tasks related to the static resources:

- **Device reference:** identify an EIP device by means of a handle; using this device handle all the static resource of the EIP device can be accessed at known offsets.
- **Resource access**: locate the EIP device static resources in the system memory map and provide 32-bit data word read and write access to the host CPU.
- **Endianness conversion**: provide a central customization point where endianness conversion can be performed efficiently when the way a data word is stored in the memory is different for the host CPU and the EIP device (see section 2.4.1.3), e.g. least or most significant byte first.
- **Tracking device access**: keep track of device access count.

The detailed specification of the Device API can be found in chapter 3.

2.4.1.1 Device Reference

Each EIP device instance is identified by a Device Reference (`Device_Handle_t`). In order to allow for an efficient implementation the Device References are provided by the `Device_Find` function. The Adapter layer can retrieve the Device Reference by means of this function and provide it to the corresponding EIP Driver Library.

2.4.1.2 Resource Access

The EIP device instance is integrated in the system hardware and the EIP Driver Library is responsible for managing access to all the static resources within the device. Usually the address offsets of the static resources and their layout are defined during EIP device design and thus are EIP device-specific. Once the EIP device is integrated in the system hardware all its static resources are mapped onto some base address. Knowing this EIP device base address and the static resources offsets, it is possible to access them from the host CPU. The EIP Driver Library is not aware of the EIP device base address as this address is specific to the system hardware. The library only contains the address offsets of the EIP device static resources.

The EIP Driver Library is also not aware of the access method to these static resources because this method depends on the way the EIP device is interfaced with the host CPU that is again specific to the system hardware. For example, the EIP device static resources can be memory mapped and accessible for the host CPU via the system bus such as Processor Local Bus (PLB) or Advanced High-performance Bus (AHB). Alternatively, the EIP device can be integrated in a PCI chip with some of its static resources mapped onto the PCI Configuration Space, which will require special access method for the host CPU. The Device API must be implemented specifically for the used system hardware in order to provide the correct access method to the EIP device static resources for the EIP Driver Library running on the host CPU.

Figure 3 shows a simple system with three EIP devices, each with their own Device Identifier. The first two EIP devices are identical and can be handled by EIP Driver Library A. The picture shows

the address offsets within each EIP device and two groups of static resources inside EIP Device #3 controlled by EIP Driver Library B. EIP Driver Library B is aware about X1 and X2 and the layout of the static resources.

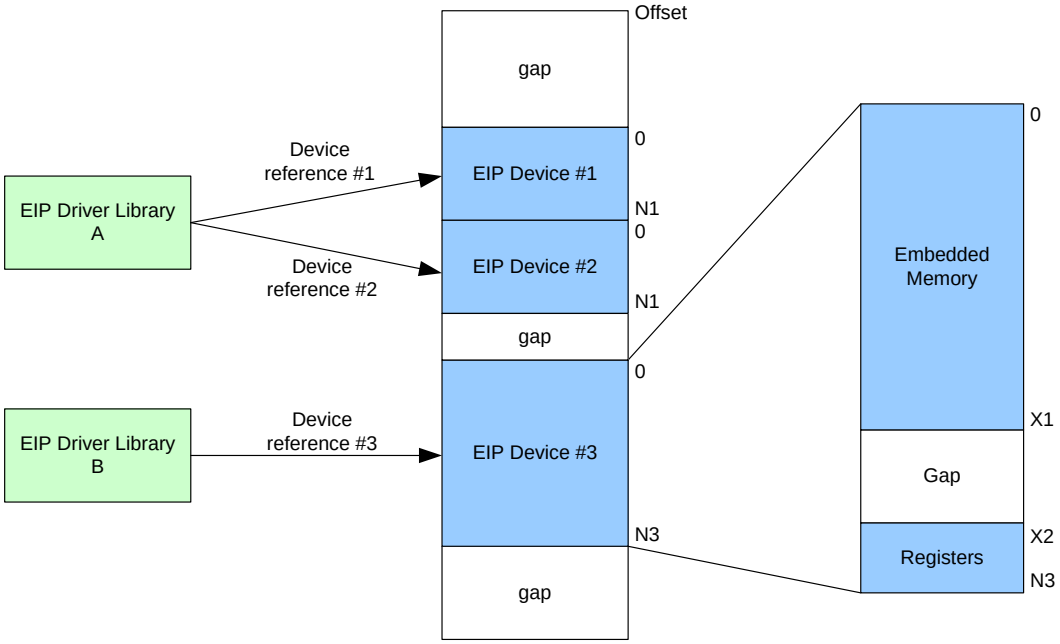

Figure 3 Static Device Resources

The EIP Driver Libraries use the `Device_Read/Write32[Array]()` functions to read/write the static resources for a specific EIP device.

2.4.1.3 Endianness Conversion

The different endianness of the host CPU and the EIP device (big-endian versus little-endian) can result in data incompatibilities that must be resolved. Some EIP devices can perform the endianness conversion run-time when the data word static resource is read or written by the host. Other devices do not support this and require the host to write the data word static resources in device-compatible endianness.

The Device API supports reading and writing 32-bit data words and allows the byte order within these data words to be swapped by the API implementation, if required. The endianness conversions functionality in the Driver Framework Device API is intended to ensure reusability of the EIP Driver Libraries across various system hardware by making them agnostic to the host CPU endianness.

It should be noted that if an EIP device supports endianness conversion in hardware then its byte-lane swapping is agnostic of resource being accessed (register or internal memory) and simply handles all data words identically. Similarly, the byte swap implementation of the Device API should do the same.

Endianness conversion must take place in both directions: when the host CPU interprets data words written by the EIP device as well as the EIP device interpreting data words written by the host CPU. Note that data processed as byte stream by the host CPU and EIP device does not require endianness conversion.

2.4.2 DMA Resource API and Dynamic Resources

The DMA Resource API of the Driver Framework must be implemented only for those EIP devices that have DMA capability. Shared buffers in the system memory can be accessed by the host CPU as well as by the DMA-capable EIP devices. These buffers are called dynamic resources in this document because they have properties opposite to those of static resources. It should be noted that dynamic resource and `DMAResource` record are different data objects. The dynamic resource is used to store data for DMA operations of the EIP device whereas the resource record is used to describe the properties of the dynamic resource.

First, an address of a dynamic resource is not known at system design time but at run-time, for example a DMA buffer can be dynamically allocated during execution of an application. Once the resource is instantiated and its address is determined, the host CPU and the EIP device can start using that address to access the resource. This property has an important consequence for the Driver Framework. Both the host CPU and EIP device may need to use different addresses to access the same dynamic resource in the system memory. This depends on whether CPU and/or I/O MMU or other address translation circuitry is used in the system hardware. Second, dynamic resources may reside in the system memory regions that are cached either by the host CPU data cache or by the EIP device. The data cache can also be present in the data path from the EIP device to the system memory.

The DMA Resource API is intended for handling these tasks related to the dynamic resources:

- **Resource identification**: identify dynamic resources by means of a record handle; the `DMAResource` record holds all the required information about a dynamic resource; the record handle is a reference to such a record.
- **Resource allocation**: allocate a dynamic resource such as a memory buffer that can safely be used for DMA operation by the EIP device and free this resource upon a request.
- **Resource access:** provide 32-bit data word read and write access for dynamic resources to the host CPU; this also includes endianness conversion – see the next item.
- **Endianness conversion**: provide a central customization point where endianness conversion can be performed for dynamic resources when the way a data word is stored in the memory is different for the host CPU and the EIP device, e.g. least or most significant byte first.
- **Data coherence handling**: provide customization points where data cache coherence can be done for dynamic resources between the host CPU and the DMA-capable EIP device.
- **Address translation**: translate address of a dynamic resource from one memory domain to another.

The detailed specification of the DMA Resource API can be found in chapter 3.

2.4.2.1 Resource Identification

The Driver Framework DMA Resource API implementation stores information about allocated dynamic resources in special `DMAResource` records. These records are also created and administered by the DMA Resource API implementation. It remains implementation-specific when to allocate the storage for these records, how many records can be supported concurrently, etc. The purpose of a record handle is to identify the corresponding dynamic resource. In an efficient implementation, the handle can be the pointer to the record. A more secure implementation can use an intermediate storage for the handle to enable detection of continued use of handles after Destroy, use a LRU recycling algorithm, etc.

The fields in the record can fully be customized to the requirements of a certain driver and system and is therefore specified as external to the API. The relation between records and handles is 1:1, although multiple records can refer to (parts of) the same shared buffer. The fields in the `DMAResource` record may be accessed directly from the Adapter implementation. Driver porting or customization may also require adding fields that are solely used in the Adapter.

Since the records can hold all the information about allocated dynamic resources, the record handles are the only piece of information to be sent around in a driver. In relation to the architecture drawing in Figure 1, the DMA Resource API helps sharing information between the Driver

Framework implementation and the Adapter layer. The EIP Driver Libraries have been designed without any dependency on fields in the resource records.

2.4.2.2 Resource Allocation

The DMA Resource API supports allocation and freeing of dynamic resources. The implementation of this functionality is platform-specific and must ensure that allocated resources can safely be used for the DMA operations by the EIP device. Moreover, this API supports registration of already allocated (outside of the driver) dynamic resources that can be re-used for DMA operations by the driver. If implemented, this feature can be used as a performance optimization method to avoid bounce buffer allocation.

2.4.2.3 Resource Access

Unlike the EIP device static resources, the dynamic resources reside in the system memory, for example off-chip SDRAM or on-chip SRAM. Although the access methods for the system memory are rather standardized, they remain specific to the system hardware and the OS. The implementation of the DMA Resource API must allow the host CPU to read and write 32-bit data words from and to these resources. This functionality can be coupled with the endianness conversion functionality (see section 2.4.2.6).

2.4.2.4 Data Coherence Handling

In case dynamic resources are allocated in the memory regions cached either by the host CPU data cache or by the EIP device, the DMA Resource API implementation must ensure data coherence. How the API implementation must do this depends on the system hardware and its data cache architecture, but in general, the following allocation schedule must be followed:

1. After the host CPU modifies (writes) the dynamic resource, but before the DMA data transfer is started from the system memory to the EIP device;
2. After the DMA data transfer is completed from the EIP device to the system memory, but before the host CPU reads that data from the dynamic resource.

2.4.2.5 Address Translation

The DMA Resource API implementation must support address translation for dynamic resources if required by the system hardware. If a dynamic resource is accessed in more than one memory domain by the host CPU executing the driver code or by the EIP device then address translation for the resource is required. For example, the API implementation may need to translate the virtual host address of a dynamic resource used by the EIP Driver Library on the host CPU to the physical address of the same dynamic resource used by the DMA engine of the EIP device.

2.4.2.6 Endianness Conversion

The DMA Resource API implementation of the endianness conversion functionality is similar to that of the Device API described in section 2.4.1.3. However, one important difference is that dynamic resources in the system memory are accessed by the EIP device via its master interface where the hardware endianness conversion can be configured, possibly, per DMA transfer type. The static resources (located inside the EIP device) are accessed by the host CPU via the slave interface of the EIP device that most probably does not support endianness conversion in hardware.

2.4.3 Configuration

The Driver Framework implementation needs configuration information such as base addresses and endianness of the EIP devices in the system's memory map, host CPU endianness. A number of options exist to provide this configuration information to the Driver Framework implementation. The following options can be considered:

- **Compile-time**, embedded in the code. Since the Driver Framework implementation is allowed to be specific for a certain system hardware and OS, the typical solution is to embed the required configuration details in the source code. Some aspects can easily be made configurable by using a configuration header file.
- **Run-time by querying the OS**. The Driver Framework implementation is allowed to use the OS services to retrieve the required configuration information. This is a typical solution for PCI systems where the OS and BIOS maintain information about the memory locations of the EIP devices integrated in PCI chips.
- **Run-time from the Adapter**. In the situation where the Adapter has the configuration information that is needed for the Driver Framework implementation, the `Device_Initialize` function can be utilized to 'download' the information. The argument can download the information in the format agreed between the Adapter and the Driver Framework implementation. It could be in textual format (even XML), a data structure, etc.
- **Run-time detection**. The Driver Framework implementation can detect the required configuration information by performing read/write accesses to the EIP device. For example, the host CPU endianness can be detected by reading a known value from a static resource in the EIP device.

2.5 Test Suite

The Driver Framework Test Suite is available to verify the Driver Framework implementation. This test suite will check the preconditions specified in this document for each API function.

Due to the flexibility allowed in the Driver Framework implementation, the test suite is highly configurable too. Each Driver Framework implementation must have a specific configuration file named `cs_dftest.h` that is used when compiling the test suite.

3 Driver Framework API Specification

This chapter describes the Driver Framework

- Basic Definitions API,
- Device API,
- DMA Resource API and
- C Library Abstraction API.

The Driver Framework must implement these API according to this specification.

Part of the API implementation may be in the header files. This allows for solutions using 'inline' constructions, pre-processor macros and customization of the `typedefs`.

The API implementation can be HW/SW platform-dependent and compiler-dependent. The Driver Framework and Adapter implementation may also use OS services.

3.1 Basic Definitions API (basic_defs.h)

This header file defines the basic types and macros as listed below. The Basic Definitions API provides decoupling from compiler specifics. Most of the specified features are part of the ISO-C99 standard.

3.1.1 uint8_t, uint16_t, uint32_t

A check should be made to guarantee that:

```
sizeof(uint8_t)  == 1
sizeof(uint16_t) == 2
sizeof(uint32_t) == 4
```

Specification:

- Specific storage size guaranteed: 1, 2 or 4 bytes.
- Unsigned data type.
- Must be possible to use as a basic type, supporting storage declaration, assignment and typecast.
- CPU-native endianness.

Example implementation:

```
typedef unsigned char  uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int   uint32_t;
```

These data types are required with exact controlled storage size of 8, 16 and 32 bits.

3.1.2 bool, true, false

These definitions are used in the driver for binary logical variables and to test for either value. Customization can be done to reach optimal performance versus storage requirement.

Specification:

- `bool` must be a basic type supporting storage declaration and assignment.
- `false` and `true` must be constants that can be assigned to a variable of type `bool`.
- `false` must be equal to zero.
- Typecast behavior to and from `bool` is undefined and should not be used.
- Only the assignment and equality-comparison operators are supported (see unsupported use below).

Note: *The ISO-C99 specification defines a more complete type that supports typecasts.*

Example implementation:

```
typedef unsigned char bool
#define false 0
#define true  (!false)
```

Examples of support use:

```
void foo(bool fBar)
{
    bool fSample = false;
    bool fSample2 = fBar;
    if (fSample2)
        n++;
    if (fBar == true)
        n++;
}
```

Examples of unsupported use:

```
int n = rand();
bool fBar = (bool)n;
int x = BIT_0 & fBar;
```

3.1.3 NULL

The definition is used to test pointers for “pointing nowhere specific” and setting pointers to represent this state.

Specification:

- `NULL` must be equal to zero. A zero-initialized data structure containing pointers is expected to have its pointers equal to `NULL` after initialization.
- Must support any pointer type.

Example implementation:

```
#define NULL 0
```

3.1.4 MIN and MAX

These two functions/macros are used to choose the lowest respectively highest value from two operands. In order to support all data types, these functions are typically implemented using pre-processor macros that expand to a construction with the ternary operator (question mark – colon construction) and a comparison operator.

Specification:

- Accepts two integer values.
- Type agnostic (must support any storage type supported by the comparison operators).
- `MIN` returns the smallest of the two values.
- `MAX` returns the largest of the two values.
- Shall not cause a function call.
- Shall not declare read/write storage (global or stack).
- Caller should avoid side-effects (do not use it with constructions like "n++").

Examples:

```
MIN(-5, 5)     → -5
MIN(0, 1) → 0
MAX(5, 100)    → 100
MAX(-10, -5)   → -5
```

Example implementation:

```
#define MIN(_x, _y) ((_x) < (_y) ? (_x) : (_y))
#define MAX(_x, _y) ((_x) > (_y) ? (_x) : (_y))
```

3.1.5 BIT_0 .. BIT_31

These pre-processor macros represent each of the bit positions 0 through 31. For optimal performance, these shall result in constant expressions after evaluation. Any typecasts (like uint32) or postfixes (like "U" or "UL") can be done here, if required.

Specification:

- `BIT_n` must evaluate to a numeric constant equaling 2^n.

Examples:

```
BIT_0          →    1
BIT_0 << 2     →    4
BIT_31         →    0x80000000
```

Example implementation:

```
#define BIT_0  0x00000001UL
#define BIT_1  0x00000002UL
#define BIT_2  0x00000004UL
…
#define BIT_30 0x40000000UL
#define BIT_31 0x80000000UL
```

Note: *The use of UL indicates Unsigned Long. This must be used with a compiler where an unsigned integer is not enough to represent a 32-bit number.*

3.1.6 MASK_n_BITS

These pre-processor macros are meant to generate mask for various bit level manipulation of a register. These can be used in cases where a subset of bits in register needs to be masked or in cases where a parameter value is to be added to the register.

Specification:

- MASK_n_BITS must evaluate to a mask in which all the lower 'n' bits are 1.

Examples:

```
MASK_1_BIT     → 0x00000001
MASK_2_BITS    → 0x00000003
MASK_21_BITS   → 0x001FFFFF
```

```
value = value & MASK_5_BITS;     // (Extract lower 5 bits from "value")
value = (value >> 6) & MASK_5_BITS; // (Extract 10:6 bits from "value")
```

Example implementation:

```
#define MASK_1_BIT  (BIT_1 - 1)
#define MASK_2_BITS (BIT_2 - 1)
#define MASK_3_BITS (BIT_3 - 1)
#define MASK_4_BITS (BIT_4 - 1)
#define MASK_5_BITS (BIT_5 - 1)
…
#define MASK_31_BITS(BIT_31 - 1)
```

3.1.7 IDENTIFIER_NOT_USED

This preprocessor macro can be used in function bodies when a function argument is not used in the body. Using this macro on that specific identifier shall avoid a compiler warning 'identifier not used'.

Specification:

- Preprocessor macro accepting identifier name.
- Shall suppress the compiler warning 'identifier not used' for that parameter.
- Must be possible to use at any position in the function body without causing side effects or warnings.
- Shall not cause a function call.
- Shall not declare read/write storage (global or stack).

Example:

```
void foo(int bar)
{
    IDENTIFIER_NOT_USED(bar);
}
```

Example implementation:

```
#define IDENTIFIER_NOT_USED(_v)  if (_v) {}
```

3.1.8 inline

After including the `basic_defs.h` header file, the keyword '`inline`' must be supported, allowing the definition of inline functions in header files. This was added because not all compilers support the inline directive and some compilers use another directive to replace 'inline'.

Specification:

- Use of the construction "static inline" must cause a function to be declared inline as intended by ISO-C99.

Examples:

```
extern int newfoo(int bar);

static inline int foo(int bar)
{
    int t = 32 + bar * 3;
    return newfoo(t);
}
```

Example implementation:

```
// definition of the inline keyword when required
// Microsoft compiler only supports "inline" in C++ mode
// and expects __inline in C mode
#ifdef _MSC_VER
#ifndef __cplusplus
#define inline __inline
#endif
#endif
```

3.2 Device API

The Device API enables the EIP Driver Library to access the registers and embedded memories of an EIP hardware block, regardless of the integration (typically bus structure, bridges, dynamic memory map, potential byte-lane swapping, etc.).

A typical embedded integration has the hardware block accessible in the memory map. The exact address might not be known at compile-time though and must be provided at run-time.

Device API overview:

- Type definitions
  - Device handle type
- Management functions
  - Initialize driver
  - Un-initialize driver
  - Find device
- Read/Write functions
  - Read or write of single 32-bit integer
  - Read or write of arrays of 32-bit integers
- Endianness conversion function
  - Swap the byte order of a 32bit data words

The device initialization function must be called prior to calling any EIP Driver Library API. The device un-initialization function must be called when no EIP Driver Libraries are used anymore. The Device API implementation should not call sleep-able OS functions (see next note).

Note: *The EIP Driver Libraries must support non-sleep-able execution contexts (like interrupt context) and therefore require the Device API implementation to not use sleep-able OS calls. If the API uses sleep-able OS calls (for example in a simulation environment) then the Adapter must guarantee that it does not call the EIP Driver Library from a non-sleep-able context.*

Additional information can be found in sections 2.2 and 2.3.

3.2.1 API States

The Device API implementation is initially in the "`UnInitialized`" state. It can be brought to the "`Device InAccessible`" state using the `Device_Initialize()` function. The EIP device is only accessible after successful completion of the `Device_Find()` and `Device_Initialize()` functions, which brings the Device API implementation for the requested device to the "`Device Accessible`" state. The transition from the "`Device InAccessible`" to the "`Device Accessible`" state can be repeated for multiple EIP devices supported by the Device API implementation. The `Device_Read32/Write32[Array]` functions can be used for the EIP device in the "`Device Accessible`" state. When the Device API implementation is not required anymore then it can be brought back to the "`UnInitialized`" state by the `Device_UnInitialize()` function.

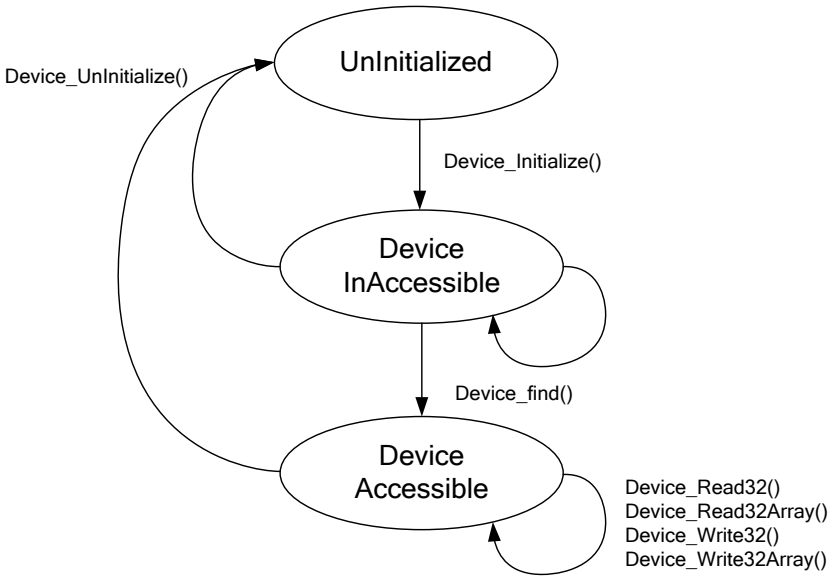

Figure 4 Device API States

3.2.2 Type Definitions (device_types.h)

3.2.2.1 Device_Handle_t

This handle represents a device, typically one EIP hardware block instance. The Device API can access the static device resources (device registers and internal memory) using offsets inside the device. This abstracts memory map knowledge and simplifies device instantiation.

Each device can have its own configuration, including the endianness conversion need for the transferred data words. Endianness conversion can thus be performed on the fly and transparent to the caller.

The details of the handle are implementation specific and must not be relied upon, with one exception: NULL is guaranteed to be a non-existing handle.

```
typedef void * Device_Handle_t;
```

3.2.2.2 Device_Reference_t

This is an implementation-specific reference to a device. It is different from the device handle in the way that the underlying implementation such as OS or hardware may be aware of this device and its reference whereas they are not aware of the device handle. The device reference can be obtained via the Device API and, for example, be used with OS services that require such a reference. For the semantic details refer to the example implementation of the DMA Resource API which uses the device reference.

The details of the device reference are implementation specific and must not be relied upon, with one exception: NULL is guaranteed to be a non-existing reference.

```
typedef void * Device_Reference_t;
```

3.2.3 Management functions (device_mgmt.h)

3.2.3.1 Device_Initialize

This function must be called exactly once to initialize the Device API implementation before any other Device API function may be used. This function brings the Device API implementation to the "Device InAccessible" state (see section 3.2.1).

```
int
Device_Initialize(
    void * CustomInitData_p)
```

Parameters:

CustomInitData_p	[in]	This anonymous parameter can be used to pass information from the caller to the Device API implementation.

Returns:

0 Success

<0 Error code (implementation specific)

>0 Reserved

3.2.3.2 Device_UnInitialize

This function can be called to shut down the Device API implementation. The caller must make sure that none of the other Device API functions are called after or during the invocation of this `Device_UnInitialize()` function. After this call, the API state is back in "`UnInitialized`" (see section 3.2.1) and the `Device_Initialize()` function may be called anew.

```
void
Device_UnInitialize(void)
```

Returns:

N/A

3.2.3.3 Device_Find

This function must be used to retrieve a handle for a certain device that is identified with a string. The supported device strings are implementation specific.
Note that this function may be called more than once to retrieve the same handle for the same device.

```
Device_Handle_t
Device_Find(
    const char * DeviceName_p)
```

Parameters:

DeviceName_p	[in]	Pointer to the (zero-terminated) string that represents the device.

Returns:

`NULL` if no device is found with the requested name.

`!NULL` the device handle that can be used in the Device API.

3.2.3.4 Device_GetReference

This function must be used to retrieve a reference for a certain device that is identified with the device handle. The format of this reference is implementation specific.
Note that this function may be called more than once to get the same reference for the same device.

```
Device_Reference_t
Device_GetReference(
    const Device_Handle_t Device)
```

Parameters:

Device	[in]	Handle for the device instance as returned by `Device_Find()`.

Returns:

`NULL` if no reference is found for the requested device handle.

`!NULL` the device reference.

3.2.4 Read/Write functions (device_rw.h)

3.2.4.1 Device_Read32

This function can be used to read one static 32-bit resource inside a device (typically a register or memory location). Since reading registers can have side effects, the implementation must guarantee that the resource will be read only once and no neighboring resources will be accessed. When the `Device` or `ByteOffset` parameters are invalid, the implementation will return an unspecified value.

```
uint32_t
Device_Read32(
    const Device_Handle_t Device,
    const unsigned int    ByteOffset)
```

Parameters:

Device	[in]	Handle for the device instance as returned by `Device_Find()`.
ByteOffset	[in]	The byte offset within the device for the resource to read.

Returns:

The value read.

Example: Read the third 32-bit register of the EIP-28 device. The registers are located back-to-back from the top of the device. The first register is at offset 0, the second at offset 4 and the third at offset 8.

```
uint32_t v = Device_Read32(DeviceHandle EIP28, 8);
```

3.2.4.2 Device_Read32Array

This function performs the same task as Device_Read32 for an array of consecutive 32-bit integers, allowing the implementation to use a more optimal burst-read (if available).

```
void
Device_Read32Array(
    const Device_Handle_t Device,
    const unsigned int    StartByteOffset,
    uint32_t *            MemoryDst_p,
    const int             Count)
```

Parameters:

Device	[in]	Handle for the device instance as returned by `Device_Find()`.
StartByteOffset	[in]	Byte offset of the first resource to read. This value is incremented by 4 for each following resource.
MemoryDst_p	[out]	Pointer to the memory where the retrieved words will be stored.
Count	[in]	The number of 32-bit words to transfer.

Returns:

N/A

3.2.4.3 Device_Write32

This function can be used to write one static 32-bit resource inside a device (typically a register or memory location). Since writing registers can have side effects, the implementation must guarantee that the resource will be written exactly once and no neighboring resources will be accessed. The write can only be successful when the `Device` and `ByteOffset` parameters are valid.

```
void
Device_Write32(
    const Device_Handle_t Device,
    const unsigned int    ByteOffset,
    const uint32_t        Value)
```

Parameters:

Device	[in]	Handle for the device instance as returned by `Device_Find()`.
ByteOffset	[in]	The byte offset within the device for the resource to write.
Value	[in]	The 32-bit value to write.

Returns:

N/A

3.2.4.4 Device_Write32Array

This function performs the same task as `Device_Write32()` for an array of consecutive 32-bit integers, allowing the implementation to use a more optimal burst-write (if available).

```
void
Device_Write32Array(
    const Device_Handle_t Device,
    const unsigned int    StartByteOffset,
    const uint32_t *      MemorySrc_p,
    const int             Count)
```

Parameters:

Device	[in]	Handle for the device instance as returned by `Device_Find()`.
StartByteOffset	[in]	Byte offset of the first resource to write. This value is incremented by 4 for each following resource.
MemorySrc_p	[out]	Pointer to the memory where the values to be written are located.
Count	[in]	The number of 32-bit integers to transfer.

Returns:

N/A

3.2.5 Endianness conversion function (device_swap.h)

3.2.5.1 Device_SwapEndian32

This function can be used to swap the byte order of a 32-bit data word. The implementation could use custom CPU instructions, if available.

```
static inline uint32_t
Device_SwapEndian32(
    const uint32_t Value)
```

Parameters:

Value	[in]	32-bit data word for byte order swap.

Returns:

The 32-bit data word with new byte order.

3.3 DMA Resource API

The DMA Resource API allows for the EIP Driver Libraries and the Adapter layer to access dynamic resources and abstracts them from the used system hardware specifics such as system bus architecture, host CPU or I/O MMU, host CPU endianness, data cache architecture.

DMA Resource API overview:

- Type definitions
  - DMA Resource handle
  - DMA Resource Record type
  - Address Domain type
  - Address Pair type
  - DMA Resource Properties type
- Management functions
  - Initialize DMA Resource administration
  - Un-Initialize DMA Resource administration
  - Create a DMA Resource record
  - Destroy a DMA Resource record
  - DMA Resource handle operations
- Read/Write functions
  - Read or write a single 32-bit integer
  - Read or write arrays of 32-bit integers
  - Data cache coherence functions
- DMA buffer management functions
  - Buffer allocate and release functions
  - Buffer register and attach functions
  - Endianness conversion control functions
- Address translation functions
  - Translate dynamic resource address
  - Register dynamic resource address

Additional information can be found in section 2.4.2.

3.3.1 API States

Initially, the DMA Resource API implementation is in the "`UnInitialized`" state. From this state the transition can be made to the "`Initialized`" state by calling the `DMAResource_Init()` function. Only in the "`Initialized`" state, the API user may call the other API functions to create resource handles and perform other operations for dynamic resources.

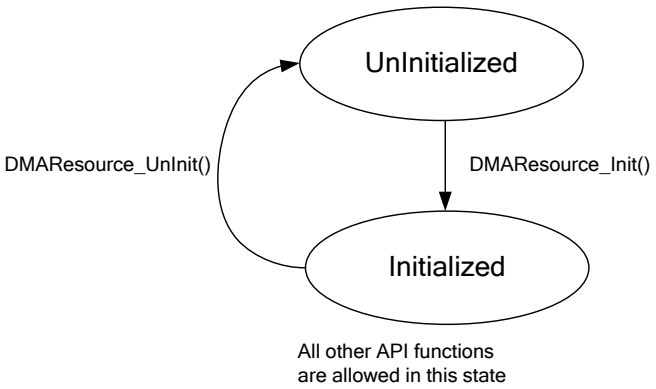

Figure 5 DMA Resource API States

Once the DMA Resource API implementation is in the "Initialized" state dynamic resources can be created via this API. Each dynamic resource created by means of this API can be described by a DMAResource record. The API defines functions for the operations with such records. Both the dynamic resource as well as the record that describes the resource can be identified by a DMAResource handle. The dynamic resource state diagram is shown in Figure 6. The dynamic resource record is instantiated by calling DMAResource_CreateRecord() function which on successful completion transits the resource from the "Unused" to the "Ready" state. The DMAResource_DestroyRecord() function in the "Ready" state can transit the resource either to the optional "Awaiting Recycle" state or directly to the "Unused" state. The "Awaiting Recycle" state is optional and it allows the API implementation to reuse destroyed records and their handles. The DMA Resource API functions taking the DMAResource handle as the input parameter may be called for a dynamic resource while it is in the "Ready" state. The following DMA Resource API functions can be used while the dynamic resource is in the "Ready" state:

- DMAResource_Read32/Write32[Array]
- DMAResource_Pre/PostDMA
- DMAResource_Translate
- DMAResource_AddPair
- DMAResource_Release
- DMAResource_SwapEndianess_Set/Get
- DMAResource_Handle2RecordPtr
- DMAResource_IsValidHandle
- DMAResource_DestroyRecord

It should be noted that in the resource "Ready" state its handle could be validated by means of the DMAResource_IsValidHandle() function before the resources is used for other operations.

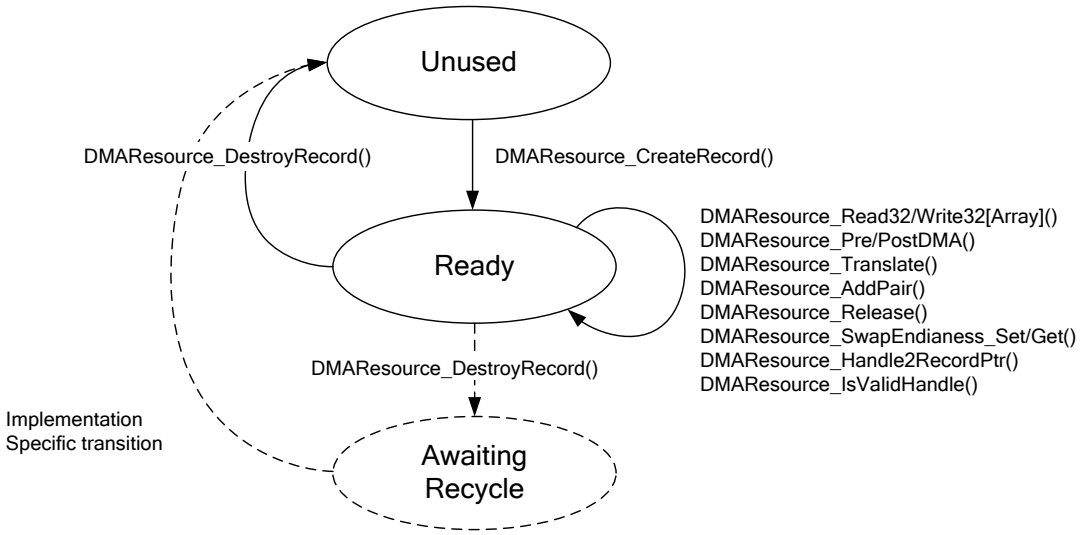

Figure 6 Dynamic Resource States

The DMAResource_CreateRecord() function can be used in the DMA Buffer Management functions DMAResource_Alloc(), DMAResource_CheckAndRegister() and DMAResource_Attach().

3.3.2 Type Definitions (dmares_types.h)

3.3.2.1 DMAResource_Handle_t

This type describes a handle that identifies a `DMAResource` record. Such a record can hold information about a dynamic resource that can be accessed by the EIP device via DMA.

```
typedef void * DMAResource_Handle_t;
```

3.3.2.2 DMAResource_AddrDomain_t

This type describes an address domain or a memory domain where a dynamic resource is used either by the host CPU or by the EIP device. One dynamic resource can be used in multiple memory domains thus requiring address translation (see section 3.3.6).

```
typedef unsigned int DMAResource_AddrDomain_t;
```

3.3.2.3 DMAResource_AddrPair_t

This type describes a dynamic resource address coupled with its memory domain. The caller is encouraged to store the address with the domain information.

```
typedef struct {
    void * Address_p;                // type ensures 64-bit support
    DMAResource_AddrDomain_t Domain;
} DMAResource_AddrPair_t;
```

3.3.2.4 DMAResource_Properties_t

This type describes dynamic resource properties. When allocating a buffer these are the requested properties for the resource. When registering or attaching to an externally allocated buffer these properties describe the already allocated buffer. The exact fields and values supported are implementation specific. For both cases, the data structure should be initialized to all zeros before filling in some or all of the fields. This ensures forward compatibility in case this structure is extended with more fields.

Example usage:

```
DMAResource_Properties_t Props = {0};
Props.fCached = true;

typedef struct
{
    uint32_t Size;       // size of the buffer in bytes
    int Alignment;       // buffer start address alignment
                         // examples: 4 for 32bit    uint8_t Bank;
                         // can be used to indicate on-chip memory
    bool fCached;        // true = SW needs to do coherency management
} DMAResource_Properties_t;
```

3.3.2.5 DMAResource_Record_t

This type describes a dynamic resource record. The format of this record is implementation specific and must be defined in a separate header file (`dmares_record.h`).

```
typedef struct
{
   // Custom Fields
} DMAResource_Record_t;
```

3.3.3 Management functions (dmares_mgmt.h)

3.3.3.1 DMAResource_Init

This function must be used to initialize the `DMAResource` administration. It must be called before any of the other `DMAResource_*` functions may be called. It may be called anew only after `DMAResource_UnInit` has been called.

```
bool
DMAResource_Init(void)
```

Returns:

`true` Initialization was successful.

`false` Initialization was not successful.

3.3.3.2 DMAResource_UnInit

This function un-initializes the `DMAResource` administration. The caller must make sure that no handles are used after this function returns. If the `DMAResource_Init()` function allocated memory, this function will free it.

```
void
DMAResource_UnInit(void)
```

Returns:

N/A

3.3.3.3 DMAResource_CreateRecord

This function creates a record for a dynamic resource and initializes the record to all zeros. The function returns a handle for the record. Use the `DMAResource_Handle2RecordPtr()` function to access the record. When the record is no longer required, it must be destroyed (see the function `DMAResource_DestroyRecord()`). The `DMAResource_CreateRecord()` function can be used by the Driver Framework implementation of the DMA Buffer Management functions `DMAResource_Alloc()`, `DMAResource_CheckAndRegister()` and `DMAResource_Attach()`.

```
DMAResource_Handle_t
DMAResource_CreateRecord(void)
```

Returns:

Handle for the dynamic resource. `NULL` is returned when the creation failed.

3.3.3.4 DMAResource_DestroyRecord

This function deletes a record. This deletes all the fields and invalidates the handle.

```
void
DMAResource_DestroyRecord(
    const DMAResource_Handle_t Handle)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).

Returns:

N/A

3.3.3.5 DMAResource_IsValidHandle

This function validates a handle.

```
bool
DMAResource_IsValidHandle(
    const DMAResource_Handle_t Handle)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).

Returns:

`true` Handle is valid.

`false` Handle is not valid.

3.3.3.6 DMAResource_Handle2RecordPtr

This function gets a pointer to the DMA Resource record (`DMAResourceRecord_t`) for the handle. The pointer is valid until the handle is destroyed with `DMAResource_DestroyRecord()`.

```
DMAResource_Record_t *
DMAResource_Handle2RecordPtr(
    const DMAResource_Handle_t Handle)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).

Returns:

Pointer to the `DMAResource_Record_t` memory for this `handle`. `NULL` is returned if the `handle` is invalid.

3.3.4 Read/Write function (dmares_rw.h)

3.3.4.1 DMAResource_Read32

This function reads a single 32-bit data word from a dynamic resource allowing the implementation to perform endianness swapping (if required). Endianness setting could be done via the custom fields in the `DMAResource` record. The implementation is able to perform run-time endianness conversion before returning to the caller. The provided resource handle and 32-bit data word offset must be valid or the function may return some unspecified value.

```
uint32_t
DMAResource_Read32(
    const DMAResource_Handle_t Handle,
    const unsigned int         WordOffset)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
WordOffset	[in]	Offset in 32-bit data words from the start of the dynamic resource to read from.

Returns:

The value read. When the `Handle` and `WordOffset` parameters are not valid, the implementation will return an unspecified value.

3.3.4.2 DMAResource_Write32

This function writes a single 32-bit data word from a dynamic resource allowing the implementation to perform endianness conversion (if required). Endianness setting could be done via the custom fields in the `DMAResource` record. The implementation is able to perform run-time endianness conversion before returning to the caller. The provided handle and 32-bit data word offset must be valid for the function to write into a correct location successfully.

```
void
DMAResource_Write32(
    const DMAResource_Handle_t Handle,
    const unsigned int         WordOffset,
    const uint32_t             Value)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
WordOffset	[in]	Offset in 32-bit data words from the start of the dynamic resource to read from.
Value	[in]	The 32-bit value to write.

Returns:

N/A

3.3.4.3 DMAResource_Read32Array

This function transfers an array of integers from a dynamic resource to another host memory location identified by the `Values_p` parameter. The implementation can call the function `DMAResource_Read32()` for a number of successive integers. The read operation can only be successful when the handle and offset parameters are valid.

```
void
DMAResource_Read32Array(
    const DMAResource_Handle_t Handle,
    const unsigned int         StartWordOffset,
    const unsigned int         WordCount,
    uint32_t *                 Values_p)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
StartWordOffset	[in]	Offset in 32-bit integers from the start of the dynamic resource to start reading from. The word offset in incremented for every word.
WordCount	[in]	The number of 32-bit words to transfer.
Values_p	[in]	Memory location to write the retrieved values to. Note that the ability to let `Values_p` point inside the `DMAResource` that is being read from, allows in-place endianness conversion.

Returns:

N/A

3.3.4.4 DMAResource_Write32Array

This function transfers an array of integers from host memory location identified by the `Values_p` parameter to the dynamic resource. The implementation can call the `DMAResource_Write32()` function for a number of successive integers. The write operation can only be successful when the handle and offset parameters are valid.

```
void
DMAResource_Write32Array(
    const DMAResource_Handle_t Handle,
    const unsigned int         StartWordOffset,
    const unsigned int         WordCount,
    const uint32_t *           Values_p)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
StartWordOffset	[in]	Offset in 32-bit words, from the start of the dynamic resource to start writing from. The word offset in incremented for every word.
WordCount	[in]	The number of 32-bit words to transfer.
Values_p	[in]	Pointer to the memory where the values to be written are located. Note that the ability to let `Values_p` point inside the `DMAResource` that is being written to, allows in-place endianness conversion.

Returns:

N/A

3.3.4.5 DMAResource_PreDMA

This function must be called when the host CPU has finished accessing the dynamic resource and before the EIP device is requested to access it via DMA. It is possible to hand off the entire dynamic resource or only a selected part of it by describing the part with a start offset and count. The implementation must use this call to ensure the data coherence of the dynamic resource between the host CPU and the EIP device.

```
void
DMAResource_PreDMA(
    const DMAResource_Handle_t Handle,
    const unsigned int         ByteOffset,
    const unsigned int         ByteCount)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
ByteOffset	[in]	Start offset within the dynamic resource for the selected part to hand off to the device.
ByteCount	[in]	Number of bytes from `ByteOffset` for the part of the dynamic resource to hand off to the device. Set to zero to hand off the entire dynamic resource.

Returns:

N/A

3.3.4.6 DMAResource_PostDMA

This function must be called after the EIP device has finished accessing the dynamic resource via DMA and before the host CPU can start accessing it. It is possible to reclaim ownership for the entire dynamic resource or only a selected part of it by describing the part with a start offset and count. The implementation must use this call to ensure the data coherence of the dynamic resource between the host CPU and the EIP device.

```
void
DMAResource_PostDMA(
    const DMAResource_Handle_t Handle,
    const unsigned int         ByteOffset,
    const unsigned int         ByteCount)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
ByteOffset	[in]	Start offset within the DMA resource for the selected part to reclaim.
ByteCount	[in]	Number of bytes from `ByteOffset` for the part of the dynamic resource to reclaim. Set to zero to reclaim the entire dynamic resource.

Returns:

N/A

3.3.5 DMA buffer management functions (dmares_buf.h)

3.3.5.1 DMAResource_Alloc

This function allocates a DMA-safe buffer of requested size that is typically used for DMA operations. The `DMAResource_CreateRecord()` function can be used by the implementation to create a record and handle to administer the buffer as a dynamic resource.

```
int
DMAResource_Alloc(
    const DMAResource_Properties_t RequestedProperties,
    DMAResource_AddrPair_t * const AddrPair_p,
    DMAResource_Handle_t * const   Handle_p)
```

Parameters:

RequestedProperties	[in]	Requested properties of the buffer that will be allocated, including the size, start address alignment, etc. See `DMAResource_Properties_t`.
AddrPair_p	[out]	Pointer to the memory location where the address and domain of the buffer will be written by this function when allocation is successful. This buffer address can be used in the caller's memory domain.
Handle_p	[out]	Pointer to the memory location where the resource handle will be returned.

Returns:

0 Success

<0 Error code (implementation specific)

>0 Reserved

3.3.5.2 DMAResource_CheckAndRegister

This function can be used to register an already allocated buffer that the caller wishes to use for DMA operations by the EIP device. The implementation can check whether the buffer is valid for the DMA operation. If this test passes then the buffer is registered. If the test fails then the buffer is rejected and not registered. The caller will then have to bounce the data to another DMA-safe buffer. The exact conditions for accepting or reject a buffer are implementation specific. The buffer must be accessible to the caller using the provided address. Use the `DMAResource_Attach()` function for buffers that are not yet accessible to the caller. It is allowed to register a subset of a DMA resource. The `DMAResource_CreateRecord()` function can be used by the implementation to create a record and handle to administer the buffer as a dynamic resource.

```
int
DMAResource_CheckAndRegister(
    const DMAResource_Properties_t ActualProperties,
    const DMAResource_AddrPair_t   AddrPair,
    const char                     AllocatorRef,
    DMAResource_Handle_t * const   Handle_p)
```

Parameters:

ActualProperties	[in]	Properties that describe the buffer that is being registered.
AddrPair	[in]	Address and Domain of the buffer. The pointer in this structure must be valid to use on the host in the domain of the caller.
AllocatorRef	[in]	Number to describe the source of this buffer. The exact numbers supported are implementation specific. This provides some flexibility for a specific implementation to provide address translation for a number of allocators.
Handle_p	[out]	Pointer to the memory location where the resource handle will be returned when registration is successful.

Returns:

0	Success
1	Rejected; buffer cannot be used for the DMA operation
<0	Error code (implementation specific)
>1	Reserved

3.3.5.3 DMAResource_Attach

This function can be used to make an already allocated buffer available in the memory domain of the caller (add it to the address map). Use this function instead of the function `DMAResource_CheckAndRegister()` when an address of the buffer is available but not for this memory domain. The exact memory domains supported by this function are implementation specific. The `DMAResource_CreateRecord()` function can be used by the implementation to create a record and handle to administer the buffer as a dynamic resource.

```
int
DMAResource_Attach(
    const DMAResource_Properties_t ActualProperties,
    const DMAResource_AddrPair_t   AddrPair,
    DMAResource_Handle_t * const   Handle_p)
```

Parameters:

ActualProperties	[in]	Properties that describe the buffer that is being registered.
AddrPair	[in]	Address and Domain of the buffer. The pointer in this structure is NOT valid to use in the domain of the caller.
Handle_p	[out]	Pointer to the memory location where the resource handle will be returned.

Returns:

0 Success

<0 Error code (implementation specific)

>0 Reserved

3.3.5.4 DMAResource_Release

This function must be used as a counter-operation for the functions `DMAResource_Alloc()`, `DMAResource_CheckAndRegister()` and `DMAResource_Attach()`. Allocated buffers are freed, attached buffers are detached and registered buffers are simply forgotten (the caller is responsible for freeing these). The caller must ensure that the buffers are not in use anymore when this function is called. The related record and handle are also destroyed using the function `DMAResource_DestroyRecord()`, so the handle and record cannot be used anymore when this function returns.

```
int
DMAResource_Release(
    const DMAResource_Handle_t Handle)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).

Returns:

0 Success

<0 Error code (implementation specific)

>0 Reserved

3.3.5.5 DMAResource_SwapEndianess_Set

This function sets the endianness conversion option in the `DMAResource` record. This field can be considered by the Read/Write functions in `dmares_rw.h` that access data words.

```
int
DMAResource_SwapEndianess_Set(
    const DMAResource_Handle_t Handle,
    const bool                 fSwapEndianess)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
fSwapEndianess	[in]	`true` = swap byte order of the data word before writing or after reading. `false` = do not swap byte order.

Returns:

0 Success

<0 Error code (implementation specific)

>0 Reserved

3.3.5.6 DMAResource_SwapEndianess_Get

This function retrieves the endianness conversion option from the `DMAResource` record. See the `DMAResource_SwapEndianess_Set()` function and section 3.3.3.6 for details.

```
int
DMAResource_SwapEndianess_Get(
    const DMAResource_Handle_t Handle)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).

Returns:

0 Success; `fSwapEndianess` is false

1 Success; `fSwapEndianess` is true

<0 Error code (implementation specific)

>1 Reserved

3.3.6 Address translation functions (dmares_addr.h)

3.3.6.1 DMAResource_Translate

This function translates the dynamic resource address already stored in the `DMAResource` record for the requested memory domain. It is typically used to get the address for the EIP device that it needs for DMA operation.

```
int
DMAResource_Translate(
    const DMAResource_Handle_t     Handle,
    const DMAResource_AddrDomain_t DestDomain,
    DMAResource_AddrPair_t * const PairOut_p)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
DestDomain	[in]	The requested domain to translate the address to.
PairOut_p	[out]	Pointer to the memory location when the converted address plus domain will be written.

Returns:

0 Success

<0 Error code (implementation dependent)

>0 Reserved

3.3.6.2 DMAResource_AddPair

This function registers another address pair known to the caller for a dynamic resource. The information can be stored in the corresponding `DMAResource` record and can be used by the `DMAResource_Translate()` function. Typically used when an external DMA buffer allocator returns two addresses (for example, virtual and physical). The number of supported address pairs is implementation specific.

```
int
DMAResource_AddPair(
    const DMAResource_Handle_t   Handle,
    const DMAResource_AddrPair_t Pair)
```

Parameters:

Handle	[in]	A valid handle that was previously returned by `DMAResource_CreateRecord()` or one of the DMA Buffer Management functions (`DMAResource_Alloc()` / `DMAResource_CheckAndRegister()` / `DMAResource_Attach()`).
Pair	[in]	Address pair (address + domain) to be associated with the DMA Resource.

Returns:

0 Success

<0 Error code (implementation dependent)

>0 Reserved

3.4 C Library Abstraction (clib.h)

The C library abstraction API enables the EIP Driver Libraries to use C library functions without including any of the header files that are provided by the compiler and might differ between compiler vendors.

Its function identifier only lists the required API. The detailed specification of the API function shall be according to the ISO-C99 standard and is not repeated here.

The following functions are required:

- `memset`
- `memcpy`
- `memmove`
- `memcmp`
- `memset`
- `offsetof`
- `strcmp`

Additional Specification:

- Non-sleep-able.
- Blocking.
- Reentrant.
- `offsetof` must be pre-processor macro that resolves at compile-time.
- Function must be safe to use in any execution context (including interrupt context) and shall not call sleep-able OS functions.

The following macro is required:

- `ZEROINIT(variable)`

This macro allows initializing the variable given as the argument with zero(s). It can be used with statically allocated arrays, data structures, variables and pointers. It will not work for dynamically allocated arrays.

Example implementation:

```
#define ZEROINIT(_x)  memset(&_x, 0, sizeof(_x))
```

Examples of support use:

```
SomeType_t SomeVar;
int SomeArray [] = {0,1,2,3};

ZEROINIT(SomeVar);
ZEROINIT(SomeArray);
```

Examples of unsupported use:

```
int * Var_p;

Var_p = malloc(size);

ZEROINIT(Var_p);
```

(End of Document)