Overview

The uVisor is a self-contained software hypervisor that creates independent secure domains on ARM Cortex-M3 and M4 micro-controllers. Its function is to increase resilience against malware and to protect secrets from leaking even among different modules of the same application.

The need for security features applies across a wide range of today's IoT products. We at ARM are convinced that many IoT security problems can be solved using standardized building blocks.

The uVisor is one of these basic building blocks – complementary to other important blocks like robust communication stacks, safe firmware updates and secure crypto-libraries.

Breaking the established flat security model of micro-controllers into compartmentalised building blocks results in high security levels, as the reach of flaws or external attacks can be limited to less sensitive function blocks.

The design philosophy of uVisor is to provide hardware-enforced compartments (sandboxes) for individual code blocks by limiting access to memories and peripherals using the existing hardware security features of the Cortex-M micro-controllers. In this documentation we refer to the sandboxes as processes.

A useful example of using uVisor features is to prevent unauthorized write access to flash memory from faulty or compromised code. This not only prevents malware from getting resident on the device, but also enables protection of device secrets like cryptographic keys.

Services built on top of our security layer can safely depend on an unclonable trusted identity, secure access to internet services and benefit from encryption key protection.

You can find more information and a **high level overview here**.

Supported platforms

The following hardware platform is currently supported for :

- **NXP FRDM-K64F**: Freedom Development Platform for Kinetis K64, K63, and K24 MCUs

The uVisor pre-linked binary images are built with the Launchpad **GCC ARM Embedded** toolchain. Currently only applications using that toolchain can be built.

High Level Design Overview

Memory Allocation System

Although many processes or secure domains might be fine with just using statically allocated memories, there's a need for reliable, safe and secure memory allocation. This need for dynamic memory allocation is difficult to satisfy on Cortex-M microcontrollers due to the lack of a Memory Management Unit (MMU).

To avoid memory fragmentation uVisor uses a nested memory allocator.

Top Level Memory Allocation

On the top level three methods exist for allocating memories for a process or thread:

- One static memory region per security context as implemented by uVisor today. The allocation happens during link time and can be influenced by compile time box configuration options. This region contains the heap and stack memories.
- After subtracting the per-process memories and the global stack, the remaining memory is split into a coarse set of equally sized large memory pages. For an instance it might make sense to split a 64kb large SRAM block into 8kb pool memory chunks.

The recommended operation for a process is to keep static memory consumption as low as possible. For occasional device operations with large dynamic memory consumption, the corresponding process temporarily allocates one or more pages of from the memory pool.

Tier-1 Page Allocator

The tier-1 page allocator hands out pages and secures access to them. It is part of the uVisor core functionality, and therefore is only accessible via the SVC-based uVisor API.

On boot, uVisor initializes the non-statically allocated heap memory into correctly aligned and equally-sized memory pages. The page size is known at compile time, however, the number of pages is known only to the allocator and only at runtime, due to alignment requirements.

Choosing a Page Size

The requested page size is passed through the uVisor input section and read by the page allocator initializer. You may overwrite the default page size of 16kB by passing the `UVISOR_PAGE_SIZE` macro with the value in bytes to the compiler.

Note, that uVisor is only able to secure up to 16 pages by default (configurable during porting). It is therefore recommended to keep the page size as large as feasible, taking into account the largest continuous memory allocation that your application requires as well as keeping the total number of available pages small.

The page size must be larger than 1kB and smaller than 512MB and must be a power-of-two to work with the ARMv7-MPU alignment restrictions.

The page allocator will verify the page size for correct alignment.

Requesting Pages

```
int uvisor_page_malloc(UvisorPageTable * const table);
```

A process can request pages by passing a page table to the allocator containing the number of required pages, the required page size as well as an array of page origins. Note that the tier-1 allocator does not allocate any memory for the page table, but works directly on the memory the user provided. If the tier-2 allocator requests 5 pages, it needs to make sure the page table is large enough to hold 5 page origins! This mechanism allows the passing statically as well as dynamically allocated page tables.

A page table structure looks like this:

```
typedef struct {
    uint32_t page_size;     /* The page size in bytes. */
    uint32_t page_count;    /* The number of pages in the page table. */
    void * page_origins[1]; /* Table of pointers to the origin of each page. */
} UvisorPageTable;
```

If enough free pages are available, the allocator will mark them as in use, zero them, and write each page origin into the page table.

The allocator returns an error code, if the page table is not formatted correctly. The requested `page_size` must be a equal to `UVISOR_PAGE_SIZE`, and all of the page table memory must be owned by the calling security context.

Note that the returned pages are **not** guaranteed to be allocated consecutively. It is the responsibility of the tier-2 allocator to make sure that memory requests for continous memory, that exceed the requested page size, are blocked.

Freeing Pages

```
int uvisor_page_free(const UvisorPageTable * const table);
```

A process can free pages by passing a page table to the allocator. The allocator first checks the validity of the page table, to make sure the pages are owned by the calling security context and then returns the pages to the pool.

Hint: Use the page table that was returned on allocation.

Tier-2 Memory Allocator

The tier-2 memory allocator provides a common interface to manage memory backed by tier-1 allocated pages or statically allocated memory. The tier-2 allocator is part of the uVisor library and calls to it does not require a secure gateway, since it can only operate on memory owned by the process anyway.

All memory management data is contained within the memory pool and its overhead depends on the management algorithm. An allocator handle is a simple opaque pointer.

```
typedef void* SecureAllocator;
```

Initializing Static Memory

Statically allocated heap memories can be initialized using this method:

```
SecureAllocator secure_allocator_create_with_pool(
    void* mem,      /**< origin address of pool */
    size_t bytes);  /**< size of pool in bytes */
```

The uVisor box heap is initialized using this method on first call to `malloc`.

Initializing Page-Backed Memory

```
SecureAllocator secure_allocator_create_with_pages(
    size_t heap_size,       /**< total heap size */
    size_t max_heap_alloc); /**< maximum continuous heap allocation */
```

The tier-2 allocator computes the required page size and page count from the `heap` size, taking account the requirement that the `max_heap_alloc` needs to be placed in one continuous memory section.

For example: With a heap size of 12kB (6kB continuous) and a physical page size of 8kB, two non-consecutive pages would satisfy this requirement. However, if instead of 6kB we would need 9kB of continuous heap allocation, we would require one page of 16kB, ie. two consecutive 8kB pages. This is not be guaranteed by the tier-1 allocator, therefore the physical page size needs to be increased to 16kB for this allocation to succeed.

Note that no guarantee can be made that the maximum continuous heap allocation will be available to the caller. This depends on the fragmentation properties of the tier-2 memory management algorithm. This computation only makes it physically possible to perform such a continuous allocation!

Note that the memory for the page table is dynamically allocated inside the process heap!

Allocator Lifetime

An allocator for static memory cannot be destroyed during program execution. Attempting to do so will result in an error.

An allocator for page-backed memory is bound to a process thread. It must not be destroyed while the thread is still running. Other threads within the same process are not allowed to allocate inside the page-backed memory of another thread. However, they may of course write and read into the page-backed memory of another thread, but that thread must first allocate memory for it and pass the pointer to the other thread.

This restriction ensures that when a thread finishes execution, it is safe to return all its pages to the memory pool, without the risk of deleting an allocation from another thread. It also removes the need to keep track which thread allocated in what page.

```
int secure_allocator_destroy(SecureAllocator allocator);
```

Memory Management

Three functions are provided to manage memory in a process:

```
void * secure_malloc(SecureAllocator allocator, size_t size);
void * secure_realloc(SecureAllocator allocator, void * ptr, size_t size);
void secure_free(SecureAllocator allocator, void * ptr);
```

These functions simply multiplex the `malloc`, `realloc` and `free` to chosen allocator. This automatically takes into account non-consecutive page tables.

The tier-2 allocator uses the CMSIS-RTOS `rt_Memory` allocator as a backend to provide thread-safe access to memory pools.

Allocator Management

The current allocator is transparently swapped out by the scheduler to provide the canonical memory management functions `malloc`, `realloc` and `free` to processes and threads. This means that calling any of these three standard memory functions, automatically uses the provided allocator. The fallback scheme used for this is “page-backed thread heap” -> “static process heap” -> “insecure global heap”:

1. In a thread with its own page-backed heap, allocations will only be services from its own heap, not the process heap. If it runs out of memory, no fallback is provided.
2. In a thread without its own page-backed heap, allocations will be serviced from the statically allocated process heap. If it runs out of memory, no fallback is provided.

3. In a process with statically allocated heap, allocations will be serviced from this heap. If it runs out of memory, no fallback is provided.
4. In a process without statically allocated heap, allocations will be serviced from the statically allocated insecure process heap.

A thread may force an allocation on the process heap using the `malloc_p`, `realloc_p` and `free_p` functions. This enables a worker thread with page-backed heap to store for example its final computation result on the process heap, and notify its completion, and then stop execution without having to wait for another thread to copy this result out of its heap.

Per-Thread Memory Allocator

All memories allocated outside of the thread will be allocated on the static heap of the process. In case a thread does set the heap pointer to NULL or the heap size to zero, memory allocations will be forwarded to the processes memory.

Starting a thread dynamically without its own heap will fallback to using the process heap:

```
/* Thread is using no dynamic memory, or allocates on the process heap. */
void * thread_stack = malloc(2kB);
if (thread_stack)
{
   osThreadDef_t thread_def;
   thread_def.stacksize = 1024;
   thread_def.stack_pointer = thread_stack;
   /* All allocations within this thread are serviced from the process heap */
   osThreadId tid = osThreadCreate(
      &thread_def,
      &task);
   /* Wait until the thread completed. */
   while( osThreadGetState(tid) != INACTIVE)
      osThreadYield();
   /* Free the stack memory. */
   free(thread_stack);
}
```

When starting a seldom-running, but high-memory-impact thread, the developer has the choice to tie the thread to a thread-specific heap:

```
SecureAllocator thread_heap = secure_allocator_create_with_pages(
   12*1024,          /* Total heap size. */
   6*1024);          /* Max. continuous heap allocation. */
if (thread_heap) {
   /* Allocate stack inside thread_heap. */
   void * thread_stack = secure_malloc(thread_heap, 1024);
   if(thread_stack) {
      osThreadDef_t thread_def;
      thread_def.stacksize = 1024;
      thread_def.stack_pointer = thread_stack;
      /* Pass the allocator to the thread. */
      osThreadId tid = osThreadCreateWithContext(
         &thread_def,
         &task,
```

```
            thread_heap);
        /* Wait until the thread completed. */
        while( osThreadGetState(tid) != INACTIVE)
            osThreadYield();
        /* Free the stack memory. */
        free(thread_stack);
    }
    /* Free the page-backed allocator. */
    secure_allocator_destroy(thread_heap);
}
```

Once the dynamic operation terminates, the threads are terminated and the corresponding memory blocks can be freed. In case allocations happened outside of the thread (using the {malloc, realloc, free}_p methods), these will be still around on the process heap.

As a result memory fragmentation can effectively avoided - independent of uVisor usage.

RTOS Integration Mechanics

As currently integrated with uVisor, the RTOS runs with uVisor privileges. This must be fixed. **This issue is tracked on GitHub at https://github.com/ARMmbed/uvisor/issues/235**. For now, the RTOS is a privileged component of the overall system, sharing privileged residence with uVisor. uVisor and the RTOS must trust each other because of this. Generally, uVisor is the manager of interrupts and the MPU, and the RTOS is the manager of context switching.

Three new capabilities are added to uVisor to support RTOS integration. - Ability for a box to specify a main thread - Ability for a privileged mode RTOS to call uVisor without an SVC - Ability for a privileged mode RTOS to specify privileged PendSV, SysTick, and SVC 0 hooks in uVisor config

Ability for a box to specify a main thread

A box's main thread is where code execution begins in that box. Each box has one main thread, with the exception of the main box (box 0). The box's "Box Config" specifies the function to use for that box's main thread.

The following is an example of box configuration. Note the use of the UVISOR_BOX_MAIN macro to specify the function to use as the body of the box's main thread.

```
/* Pre-declaration of box main thread function */
static void example_box_main(const void *);

/* Box configuration */
UVISOR_BOX_NAMESPACE(NULL);
UVISOR_BOX_HEAPSIZE(8192);
UVISOR_BOX_MAIN(example_box_main, osPriorityNormal);
UVISOR_BOX_CONFIG(example_box, acl, UVISOR_BOX_STACK_SIZE, box_context);
```

The main box's code execution starts the same way that it starts when uVisor is not present.

For all boxes other than the main box: after libc, the RTOS, and C++ statically constructed objects are initialized, and the RTOS is about to start, the RTOS asks uVisor to handle the "pre-start" event. uVisor then creates main threads for each box. The main threads use the box stack as their stack. These threads do not start until after the RTOS scheduler starts (which is after pre-start is finished).

Ability for a privileged mode RTOS to call uVisor without an SVC

A privileged call mechanism (privcall) is introduced that allows privileged code to call into uVisor without performing an SVC. This mechanism is necessary because an SVC handler can't perform another SVC (unless it first deprivileges along the way, but that would be inefficient). The only client of the privcall interface is the RTOS, which is trusted.

The RTOS needs to call uVisor to perform the following tasks: - Notify uVisor of thread creation (`thread_create`) - uVisor uses this to track which thread belongs to which box. The box that is active when a thread is created is the box that owns that thread. - Notify uVisor of thread destruction (`thread_destroy`) - uVisor uses this to forget about threads it doesn't need to track anymore. - Notify uVisor of thread switching (`thread_switch`) - In this privcall, uVisor switches the box context to the owner of the thread. - Notify uVisor that the RTOS is about to start (`pre_start`) - uVisor will create the main threads for all boxes

Ability for a privileged mode RTOS to specify privileged PendSV, SysTick, and SVC 0 hooks in uVisor config

uVisor config is stored in flash, which is trusted. As such, it's the best place for privileged subsystems, like an RTOS, to register privileged handlers. uVisor config is extended to allow the specification of the following handlers via "Privileged system IRQ hooks".

The Privileged system IRQ hooks can be used to specify the following handlers: - PendSV - RTX would register for handling PendSV to perform thread context switching - SysTick - RTX would register for handling SysTick (if a better periodic timer suitable for the RTX scheduler isn't available) - SVC 0 - RTX would register for handling SVC 0, with which RTX handles its own syscalls

Remote Procedure Calls

uVisor provides a Remote Procedure Call (RPC) API to call functions that execute in the context of another box.

A General Overview of the RPC API

The RPC API provides a structured way for a caller box to perform actions in a callee box. By default, boxes can't call functions in other box's contexts. A callee box declares RPC gateways to designate functions as callable by other boxes.

uVisor strictly controls the information passed between boxes, verifying that the callee box is OK with being called. This verification is done via an RPC gateway. An RPC gateway is a verifiable, in-flash data structure that the callee box uses to nominate a function as a suitable RPC target.

If an RPC gateway exists in a callee box, then any other box can call that target function in callee box context. No other target functions can be called in a callee box other than those designated as callable via an RPC gateway.

RPC Macros

Two macros are provided to implement RPC gateways: `UVISOR_BOX_RPC_GATEWAY_SYNC` and `UVISOR_BOX_RPC_GATEWAY_ASYNC`. - `UVISOR_BOX_RPC_GATEWAY_SYNC` creates a callable *synchronous* RPC gateway - `UVISOR_BOX_RPC_GATEWAY_ASYNC` creates a callable *asynchronous* RPC gateway

RPC gateways can be created for any function that accepts up to four, 4-byte parameters and returns up to one 4-byte value.

Calling a synchronous RPC gateway is simple. A synchronous RPC gateway is called in the same manner that the original target function would have been called. `UVISOR_BOX_RPC_GATEWAY_SYNC` creates a gateway with an identical function signature.

Calling an asynchronous RPC gateway is a bit more involved, as `UVISOR_BOX_RPC_GATEWAY_ASYNC` creates a gateway with a different function signature. Compared to the original target function, the return type is changed. The return type of the gateway is a token that can be used to wait for the asynchronous call. Sometimes this token may be invalid, in cases where the asynchronous call couldn't be initiated.

Porting a library to uVisor

To enable uVisor for a pre-existing library: 1. Make a box configuration file, `secure_libraryname.cpp` 1. Configure the box 1. Create secure RPC gateways to designate library functions as securely callable within the newly created box 1. Write incoming RPC handlers for all RPC target functions

Creating a secure gateway for a library function

We'll now work through a short example, creating both a synchronous RPC gateway and an asynchronous RPC gateway for a single library function.

Here is the imaginary function we want to make callable through RPC.

```
/* unicorn.h */
typedef enum {
    UNICORN_BARFABLE_NOTHING = 0,
    UNICORN_BARFABLE_GRASS,
    UNICORN_BARFABLE_WEEDS,
    UNICORN_BARFABLE_FLOWERS,
    UNICORN_BARFABLE_RAINBOW,
} unicorn_barfable_t;

void unicorn_barf(unicorn_barfable_t thing);
```

The function causes a unicorn to barf up some barfable thing (imaginarily, of course).

Creating a synchronous RPC gateway

To make a synchronous RPC gateway, we use the `UVISOR_BOX_RPC_GATEWAY_SYNC` macro.

```
#define UVISOR_BOX_RPC_GATEWAY_SYNC(box_name, gw_name, fn_name, fn_ret, ...)
```

These are the parameters. * `box_name` - The name of the target box as declared in `UVISOR_BOX_CONFIG` * `gw_name` - The new, callable function pointer for performing RPC * `fn_name` - The function being designated as an RPC target * `fn_ret` - The return type of the function being designated as an RPC target * `__VA_ARGS__` - The type of each parameter passed to the target function. There can be up to 4 parameters in a target function. Each parameter must be no more than uint32_t in size. If the target function accepts no arguments, pass `void` here.

Here's how to designate the function as a synchronously callable RPC target.

```
/* secure_unicorn.cpp */
#include "unicorn.h"

UVISOR_BOX_RPC_GATEWAY_SYNC(unicorn_box, unicorn_barf_sync, unicorn_barf, void, unicorn_barfable_t);
```

We also need to declare the gateway's function prototype, so that clients can call the freshly-created RPC gateway. Notice that the gateway creating macro made a function pointer and not a function, so we declare

the gateway's function prototype as a function pointer. We also need to extern the function pointer, to let the compiler know that the gateway creating macro already created the function pointer for us.

```
/* secure_unicorn.h */
UVISOR_EXTERN void (*unicorn_barf_sync)(unicorn_barfable_t thing);
```

Creating an asynchronous RPC gateway

Creating the asynchronous RPC gateway is just about as easy as creating an synchronous gateway.

To make an asynchronous RPC gateway, we use the `UVISOR_BOX_RPC_GATEWAY_ASYNC` macro.

```
#define UVISOR_BOX_RPC_GATEWAY_ASYNC(box_name, gw_name, fn_name, fn_ret, ...)
```

The parameters are the same as the `UVISOR_BOX_RPC_GATEWAY_SYNC` macro. * `box_name` - The name of the target box as declared in `UVISOR_BOX_CONFIG` * `gw_name` - The new, callable function pointer for performing RPC * `fn_name` - The function being designated as an RPC target * `fn_ret` - The return type of the function being designated as an RPC target * `__VA_ARGS__` - The type of each parameter passed to the target function. There can be up to 4 parameters in a target function. Each parameter must be no more than uint32_t in size. If the target function accepts no arguments, pass `void` here.

Here's how to make the function an asynchronously callable RPC target.

```
/* secure_unicorn.cpp */
#include "unicorn.h"

/* Both of these declarations are independent; one is not necessary for the
 * other. Both declarations are listed here for illustrative purposes only. */
UVISOR_BOX_RPC_GATEWAY_SYNC(unicorn_box, unicorn_barf_sync, unicorn_barf, void, unicorn_barfable_t);
UVISOR_BOX_RPC_GATEWAY_ASYNC(unicorn_box, unicorn_barf_async, unicorn_barf, void, unicorn_barfable_t);
```

Just like in the synchronous case, we again declare the the gateway's function prototype. This time, however, notice that the return value is of type `uvisor_rpc_result_t`. The return value is a token that facilitates the asynchronous calling of our gateway.

```
/* secure_unicorn.h */
UVISOR_EXTERN void (*unicorn_barf_sync)(unicorn_barfable_t thing);
UVISOR_EXTERN uvisor_result_t (*unicorn_barf_async)(unicorn_barfable_t thing);
```

That's all there is to it. That's all it takes to create an RPC gateway to a target function.

Handling incoming RPC

Each box has a single queue for handling incoming RPC calls. uVisor will verify the secure RPC gateways and then place calls into the target box's queue; an RPC call won't be added to the queue if the gateway isn't valid.

Making a box capable of handling incoming RPC requires two steps. 1. Specify the maximum number of incoming RPC calls for the box 1. Call `rpc_fncall_waitfor` from at least one thread

Limiting the maximum number of incoming RPC calls

To specify the maximum number of incoming RPC calls for a box, the following macro is used.

```
UVISOR_BOX_RPC_MAX_INCOMING(max_num_incoming_rpc)
```

Before the box configuration, use the `UVISOR_BOX_RPC_MAX_INCOMING` macro to specify how many RPC calls can be queued up at once.

```
/* secure_unicorn.cpp */

UVISOR_BOX_RPC_MAX_INCOMING(10);
```

With the above configuration, up to 10 RPC calls can be queued up for all RPC executors. If the executors can't execute incoming RPC calls fast enough, uVisor will prevent new RPC calls from getting queued up until space allows.

So, what happens on the caller side when a callee can't handle their call? Asynchronous callers will receive a timeout if the call can't be completed quickly enough. Synchronous callers will block forever until space is available.

Executing RPC calls

An RPC needs some context in which to execute. The context in which an RPC runs is designated by a call to `rpc_fncall_waitfor`. This function handles RPC calls, and then performs the RPC within its context. `rpc_fncall_waitfor` will either execute one RPC before returning or return with a status code indicating that something else happened.

Let's have a look at this function.

```
int rpc_fncall_waitfor(const TFN_Ptr fn_ptr_array[], size_t fn_count, uint32_t timeout_ms);
```

There are not so many parameters. * `fn_ptr_array` - an array of RPC function targets that this call to `rpc_fncall_waitfor` should handle RPC to * `fn_count` - the number of function targets in this array * `timeout_ms` - specifies how long to wait (in ms) for an incoming RPC calls before returning

And finally, the return value specifies the status of the wait (whether it timed out, or if the pool is too small, or if an RPC was handled).

Let's see what this would look like in practice for the unicorn library. Let's implement the body of a new thread to run in `unicorn_box` that will be used to handle RPC to the `unicorn_barf` target function. We'll have it wait forever for an incoming RPC call, handle it, and then wait for the next item.

```
static void unicorn_barf_rpc_thread(const void *)
{
  /* The list of functions we are interested in handling RPC requests for */
  const TFN_Ptr my_fn_array[] = {
    (TFN_Ptr) unicorn_barf, // Note use of `unicorn_barf`, not `unicorn_barf_async`
  };

  while (1) {
    int status;
    static const uint32_t timeout_ms = UVISOR_WAIT_FOREVER;

    status = rpc_fncall_waitfor(my_fn_array, ARRAY_COUNT(my_fn_array), timeout_ms);
    if (!status) {
      /* ... Handle unsuccessful status ... */
    }
  }
}
```

If we wanted to handle incoming RPC calls for additional RPC targets in this same thread, we could add them to `my_fn_array`. Also, if we so desired, we could create multiple threads to wait for the same RPC targets, as might be useful in a multi-core system to handle incoming RPC calls in parallel.

So, that about sums it up for library authors. Get out there and uVisor-enable your libraries.

Next up is a description of how to call these gateways.

Calling a secure gateway

Continuing with our previous example, we'll now work through how to call both a synchronous RPC gateway and an asynchronous RPC gateway.

Calling a synchronous RPC gateway

As a convenient reminder, this is the function prototype of the synchronous RPC gateway we created in the previous section.

```
/* secure_unicorn.h */
UVISOR_EXTERN void (*unicorn_barf_sync)(unicorn_barfable_t thing);
```

Calling this synchronous RPC gateway is really easy. The call looks exactly like a non-RPC to the target function. Ready?

```
/* example.cpp */
/* ... */
void example_sync(void)
{
    unicorn_barf_sync(UNICORN_BARFABLE_RAINBOW);
}
```

Yup, that's it. That's all there is. The call will block until the target function executes and returns. If you want to only wait for a certain amount of time for the target function to return, then you'll want to use the asynchronous RPC gateway (which we'll conveniently cover right now.)

Calling an asynchronous RPC gateway

As another convenient reminder, this is the function prototype of the asynchronous RPC gateway we created in the previous section.

```
/* secure_unicorn.h */
UVISOR_EXTERN uvisor_result_t (*unicorn_barf_async)(unicorn_barfable_t thing);
```

Now, to make the call. This isn't so different from the synchronous case, but we don't yet get the return value of the target function by calling an asynchronous RPC gateway. We instead get a `uvisor_rpc_result_t` token.

```
/* example.cpp */
/* ... */
void example(void)
{
    uvisor_rpc_result_t result;

    result = unicorn_barf_async(UNICORN_BARFABLE_RAINBOW);
    if (result == UVISOR_INVALID_RESULT) {
        /* The asynchronous call failed. */
    }

    /* ... Do anything asynchronously here ... */

    /* ... */
```

Now the asynchronous call is initiated and we are free to do stuff asynchronously. Eventually, we'll want to wait for the call to complete. Before we get to actually waiting for the call to complete, let's introduce the function that will do the waiting for us.

```
int rpc_fncall_wait(uvisor_result_t result, uint32_t timeout_ms, uint32_t * ret);
```

To use `rpc_fncall_wait`, pass in: * `result` - the result token previously received from an asynchronous call * `timeout_ms` - a timeout in milliseconds of how long to wait for a result to come back from the RPC target function * `ret` - a pointer to a `uint32_t`-sized return value

In our case, `unicorn_barf` has a void return value, so we can pass in `NULL` for `ret`.

Now that we understand how to use `rpc_fncall_wait`, let's spin in a loop, waiting for the result to come back for up to 500 ms. If we don't get a result by then, we can consider the unicorn to have ran out of barf. Any unicorn worth their salt should be able to barf within 500 ms.

```
/* example.cpp */
/* ... */
void example(void)
{
  uvisor_rpc_result_t result;

  result = unicorn_barf_async(UNICORN_BARFABLE_RAINBOW);
  if (result == UVISOR_INVALID_RESULT) {
    /* The asynchronous call failed. */
  }

  /* ... Do anything asynchronously here ... */

  /* Wait for a non-error result synchronously.
   * Note that this wait could potentially be from a different thread. */
  while (1) {
    int status;
    static const uint32_t timeout_ms = 500;

    status = rpc_fncall_wait(&result, timeout_ms, NULL);
    if (!status) {
      break;
    }
  }
}
```

Calling the asynchronous RPC gateway is as tough as it gets, and it isn't really that bad, is it?

That about wraps it up for the RPC API. We've covered both how to uVisor-enable a library and how to use a uVisor-enabled library.

More Information on the RPC API

For more information on the RPC API, please refer to **the well-commented RPC API C header file which is available at https://github.com/ARMmbed/uvisor/blob/master/api/inc/rpc.h**.

Appendix

Secure Boot & uVisor Integration

The uVisor is initialized right after device reset. For allowing application of System-On-Chip (SoC) specific quirks or clock initialization a the SystemInit hook is available for early hardware clock initialization. After

initializing the *A*ccess *C*ontrol *L*ists (ACL's) for each individual security domain (process) the uVisor sets up a protected environment using a Memory Protection Unit (the ARM Cortex-M MPU or a vendor-specific alternative).

After initializing itself, the uVisor turns the memory protection turned on, drops execution to unprivileged mode and starts the operating system initialization and continues the boot process.

In case of privileged interrupts, uVisor forwards and de-privileges them and passes execution to the unprivileged operating system or applications interrupt handler. To protect from information leakage between mutually distrustful security domains, uVisor saves and clears all relevant CPU core registers when interrupting one security context by another registers leakage when switching execution between privileged and unprivileged code and between mutually untrusted unprivileged modules.

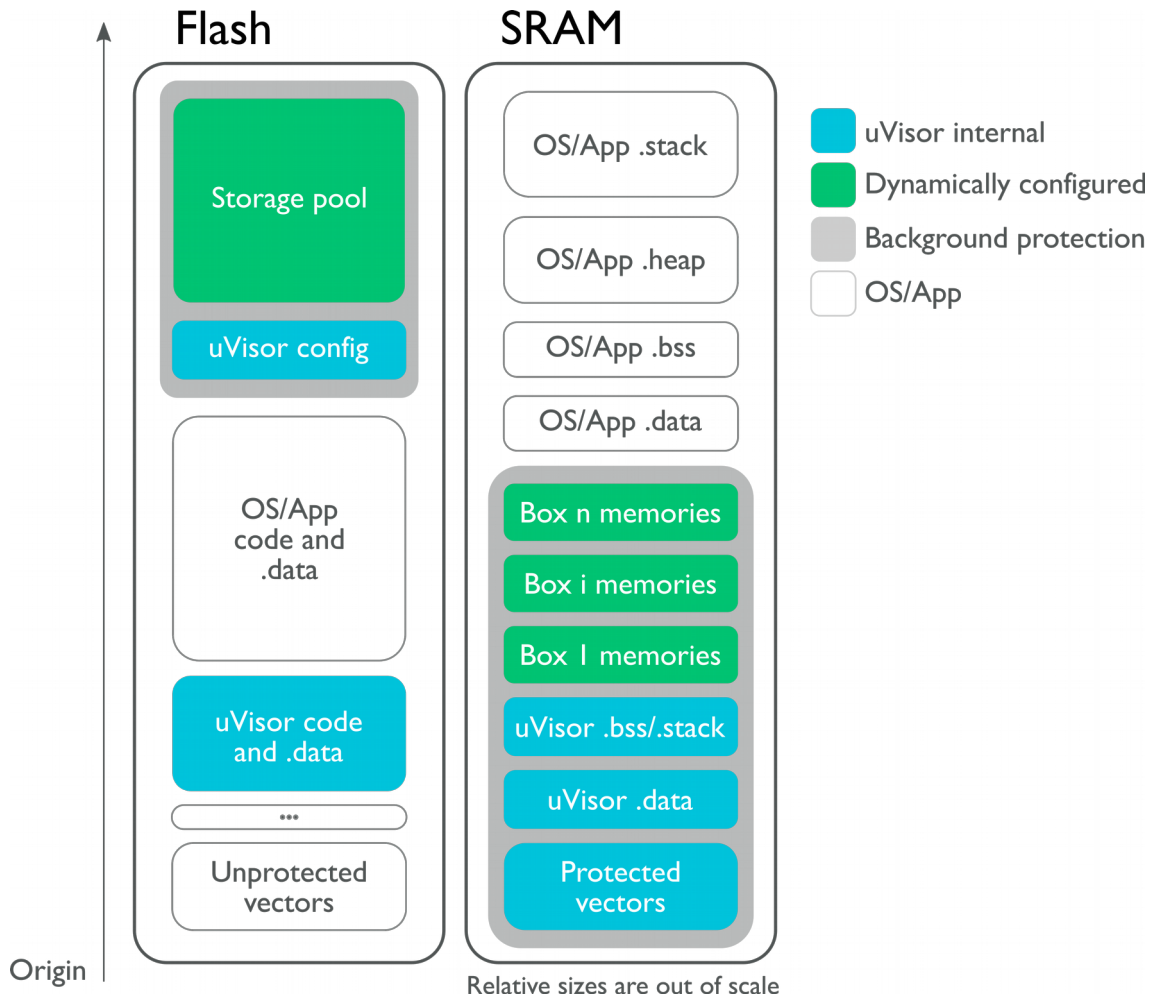
Memory access via DMA engines can potentially bypass uVisor security policies - therefore access to security-critical peripherals (like DMA) need to be restricted to permitted memory ranges by SVCall-based APIs.

Memory Layout and Management

Different memory layouts can be used on different platforms, depending on the implemented memory protection scheme and on the MPU architecture. The following figure shows the memory layout of a system where the uVisor shares the SRAM module with the operating system (ARMv7-M MPU).

uVisor provides only methods for allocating and de-allocating large blocks of uniformely sized memory to avoid fragmentation (for example: 4kb chunk size on a 64kb sized SRAM).

Each process gets at compile-time a fixed memory pool assigned which will be used for heap, stack, thread-local storage and messaging.



The uVisor secures two main memory blocks, in flash and SRAM respectively. In both cases, it protects its own data and the data of the secure boxes it manages for the unprivileged code.

All the unprivileged code that is not protected in a secure domain is referred to as the *main process*.

The main memory sections protected by uVisor are detailed in the following table:

Memory section	Description
uVisor code	The uVisor code is readable and executable by unprivileged code, so that code sharing is facilitated and privileged-unprivileged transitions are easier.
uVisor data/BSS/stack	The uVisor places all its constants, initialized and uninitialized data and the stack in secured areas of memory, separated from the unprivileged code.
Secure boxes data/BSS/stack	Through a configuration process, unprivileged code can set up a secure process for which data and stack can be secured by the uVisor and placed in isolated and protected memory areas.
Vector table	Interrupt vectors are relocated to the SRAM but protected by the uVisor. Access to IRQs is made through specific APIs.