Tame the (q)emu: debug firmware on custom emulated board

Damien «virtualabs» Cauquil , Quarkslab
dcauquil@quarkslab.com

February 1, 2024

Abstract

QEMU is one of the most used software to perform efficient executable files and systems emulation, and inspired multiple tools like `Avatar2`[1], `Panda`[11] or the `Unicorn Engine`[8] using CPU emulation for security research and training. Emulating a computer or an embedded system with QEMU is quite straightforward and documented, but emulating a board based on a microcontroller or a system-on-chip is a different story.

Therefore, modifying QEMU to emulate a specific target system is sometimes the only option regarding performances and other benefits QEMU provides, but is often seen by security researchers or trainers as very difficult or impossible to do because of the complexity of QEMU.

In this paper, we first briefly explain the main core concepts of QEMU including some of its internals and the QEMU Object Model. Then, we demonstrate that adding a custom board in QEMU is not a tedious task and can be done with little knowledge of its API, based on a specific board we use in trainings and hardware CTFs. Finally, we quickly demonstrate how this custom emulated board can be used for dynamic analysis and vulnerability research using QEMU debugging capabilities.

1 Introduction

In cybersecurity, emulating a system is mostly used for two different purposes: security assessment and offensive security training. Emulation offers a lot of advantages:

- You have complete control over the hardware: if something goes wrong you will not brick or destroy a real (possibly expensive) device.
- You can overcome some limitations imposed by the system, giving you better control over the software that runs in the emulated environment.
- You can closely monitor what is happening on the target system.

These advantages obviously make the life of the security evaluator or of the trainer easier, by allowing the use of well-known tools to be introduced in the target system while the device does not allow its firmware to be modified for instance, or by allowing kernel debugging where the target device does not offer any debug port. Trainings based on emulated hardware devices allow students to break anything and start over with a fresh new device, in a matter of seconds, without destroying real hardware. Moreover, it allows the trainer to rely on a frozen version of the firmware without any fear of a firmware being updated by the device vendor in a more recent version of a target device (true fact, it happened to the author of this submission a few years ago).

Finally, it does also apply to bare-metal systems based on microcontrollers or system-on-chips, as emulation opens up a lot of possibilities for security testing and device debugging.

1.1 Why emulating devices with QEMU?

When it comes to CPU and system emulation, QEMU is the reference software that comes into mind for many reasons: it can emulate both executable files and operating systems on emulated CPUs and hardware, it supports a wide variety of CPU and hardware, and is completely open-source. This is the go-to solution for emulation that has inspired Google's Android Emulator[3], Renesas High-Speed Simulator for R-Car[10], or even Unicorn Engine[8] or Avatar2[1].

Indeed, it is the main tool used for firmware emulation and debugging by security researchers, as it provides a convenient way to emulate embedded Linux systems among others. As an example, we presented in 2021 an instrumentation framework at the Pass The Salt conference[2] that relies on QEMU, inspired by previous work by Saumil Shah on ARMX (now EMUX[9]).

This works pretty well to emulate a real computer and run any operating system, but emulating a simple board based on a microcontroller that runs a simple firmware with QEMU is more difficult as QEMU supports a limited number of boards. For this specific use-case, we must often consider using a different tool such as the Unicorn Engine and sacrifice performances to emulate a specific microcontroller with its main hardware peripherals and other board components. But what does it cost to improve QEMU to support a custom board?

1.2 State of the Art

QEMU is a complex software but its internals has been really well documented on a dedicated blog[4] maintained by Airbus Security Lab (see fig. 1) and also in the official documentation[7].

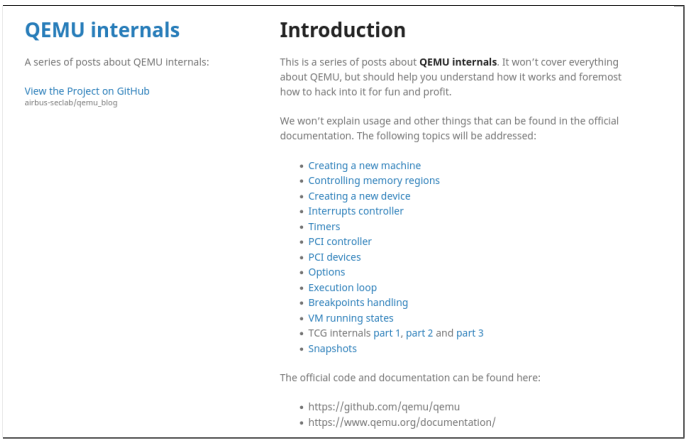

Figure 1: Screenshot of Airbus Security Lab's QEMU internals blog

However, it mostly focuses on documenting the QEMU internals and provides only a set of code snippets that are useful for whoever wants to play with QEMU, but not totally as no complete example source code is provided. Moreover, reading QEMU source code does not provide much information as most of the different operations supported by any specific component are not explained. It is difficult for a new-comer to understand what a piece of code does, and even more difficult **to understand why it has been implemented this way**.

1.3 Contribution

This is why we propose in this paper to draw a big picture of how QEMU software has been thought, its main principles and components, and a guide on how to add a simple microcontroller-based custom board with the corresponding fully documented source code.

The source code will be released soon before SSTIC and will be made available on Github.

2 QEMU software architecture

2.1 The QEMU Object Model

QEMU is developed in C but the developers created a specific code architecture that allows QEMU to provide some sort of *pseudo-classes* that can be instanciated at run-time, these *pseudo-classes* being the basic building blocks of an emulated system. For simplicity, we will call them *classes* for the rest of the paper even if they are not real classes.

Each supported CPU is a *class* that derives from a generic CPU class, and this is the same for all emulated hardware peripherals, system-on-chips, boards and machines. These classes allows a great modularity, and each derived class must implement a set of functions to provide the expected behavior. These classes compose the *QEMU Object Model (QOM)*.

From a technical perspective, those classes are basically defined as C structures like the following:

```
typedef struct M25P80Class {
    SSISlaveClass parent_class;
    FlashPartInfo *pi;
} M25P80Class;
```

In this example, the `SSISlaveClass` structure defines the base class with its properties and methods:

```
/* Slave devices.  */
struct SSISlaveClass {
    DeviceClass parent_class;

    void (*realize)(SSISlave *dev, Error **errp);

    /* if you have standard or no CS behaviour, just override transfer.
    * This is called when the device cs is active (true by default).
    */
    uint32_t (*transfer)(SSISlave *dev, uint32_t val);
    /* called when the CS line changes. Optional, devices only need to implement
    * this if they have side effects associated with the cs line (beyond
    * tristating the txrx lines).
    */
    int (*set_cs)(SSISlave *dev, bool select);
    /* define whether or not CS exists and is active low/high */
    SSICSMode cs_polarity;

    /* if you have non-standard CS behaviour override this to take control
    * of the CS behaviour at the device level. transfer, set_cs, and
    * cs_polarity are unused if this is overwritten. Transfer_raw will
    * always be called for the device for every txrx access to the parent bus
    */
    uint32_t (*transfer_raw)(SSISlave *dev, uint32_t val);
};
```

And of course this `SSISlaveClass` structure inherits from the base device class, defined with its own structure:

```
typedef struct DeviceClass {
    /*< private >*/
    ObjectClass parent_class;
    /*< public >*/

    // [...]

    bool user_creatable;
    bool hotpluggable;

    /* callbacks */
    DeviceReset reset;
    DeviceRealize realize;
    DeviceUnrealize unrealize;

    /* device state */
    const VMStateDescription *vmsd;

    /* Private to qdev / bus.  */
    const char *bus_type;
} DeviceClass;
```

This method based on nested structures allows generic APIs to manipulate the various fields of a defined class that inherits from a base class, while keeping the APIs manipulating the base class functional. **This mechanism is the core of the QEMU Object Model**.

Defining a new hardware peripheral or a new SPI flash memory chip is basically done the same way: we first need to define a dedicated structure associated to the emulated component with its first member declaring the base class structure, and then add our own fields after. Each time QEMU is asked to create an object of this specific type, it will allocate the corresponding structure, initialize it by calling some specific functions and then let the associated code handle everything.

Moreover, each class type is given a unique name that is used to reference it by other building blocks, and QEMU provides all the required primitives to instanciate an object based on its class name as well as access its properties from outside its implementation, this is covered later in this paper (see 3.4).

2.2 Assembling components like Lego bricks

Considering each emulated component as an object class be it a CPU, a hardware peripheral, a specific bus or even a system-on-chip, allows anyone to define a machine or a board by simply instanciating and connecting objects like Lego bricks. This is the true power of QEMU: we can reuse any

component if it is already supported!

Communication between the different instanciated objects that form a machine or a board is performed through two different types of communication channel: system buses and `GPIO`s (`General Purpose Input/Output`). System buses mostly represent a classic bus used on real hardware, that supports one or multiple peripherals connected to it as well as bi-directionnal data transfer. `GPIO`s however allow to connect some outputs from a QEMU object to the input of another one, similarly to real hardware devices. These `GPIO`s are then handled by QEMU, and can be used to propagate an interrupt request (`IRQ`) or a physical state of an `IO` for other purposes.

QEMU objects assemblies are also defined as QEMU object classes, with their own structures, names and specific callbacks to handle initialization, instanciation and destruction of their objects.

This section will be more detailed in the final article, including examples of basic blocks that can be assembled and more information on system buses and GPIOs.

2.3 QEMU source code tree

The QEMU source code tree is quite huge, but there is some logic behind it. This section details the most important sections, what they contain and what role they play in QEMU software architecture.

2.3.1 /hw

This folder is one of the most important regarding our purpose, as it is where all hardware devices emulation code is stored. Subfolders include:

- `/hw/arm` containing every supported ARM-based system-on-chips and boards.
- `/hw/block` containing block device drivers (usually devices that provides storage capabilities like Flash memory chips).
- `/hw/char` containing character device drivers, mostly serial devices (UART).
- `/hw/i2c` containing various i2c devices implementation.
- `/hw/intc` containing different interrupt controllers.
- `/hw/misc` containing different controllers and devices that do not fit the other categories.
- `/hw/ssi` containing various SPI hardware peripherals implementation.

2.3.2 /qom and /qobject

These folders provide the main code for the QEMU Object Model and QEMU Object. This code should not be modified as it is part of the core code of QEMU. However, it can be interesting if you want to understand how the QEMU Object Model and objects are managed by QEMU.

2.3.3 /target

This folder provides the various implementations of target CPUs. Again, this is not a part of QEMU source code we need to modify, but it may be of interest if you need some information regarding how a specific CPU is supported or the different variants it can accept.

3 Adding a custom board in QEMU

3.1 Introducing Quarkslab's Lil'Board

Quarkslab designed and produced a tiny board to be used in trainings and its Hardware Capture the Flag events, the *Lil'Board*. This board is based upon a tiny but powerful Microchip SAMD21 microcontroller, an external SPI flash chip and a simple power circuit based on a low-dropout voltage regulator. The board exposes a number of pin headers connected to the main components.

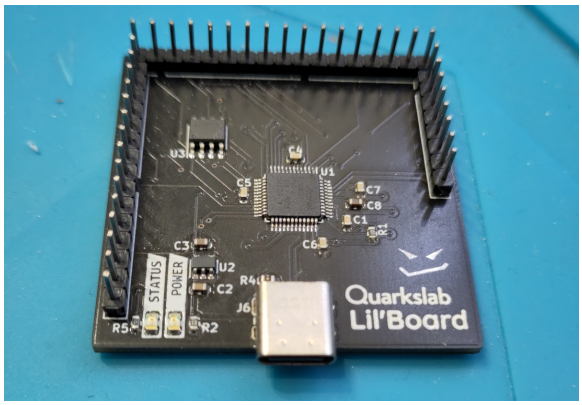

Figure 2: Quarkslab Lil'board

The SAMD21 microcontroller does not require an external oscillator but only some decoupling capacitors, and is still able to communicate over USB thanks to its USB clock recovery feature. This makes the board very simple to analyze and emulate.

3.2 Adding support for a specific microcontroller (ATSAMD21)

The SAMD21 microcontroller is based on a Cortex-M0+ ARM CPU that is supported out-of-the-box by QEMU. Most of its hardware peripherals are however not supported by QEMU and we need to add some dedicated code to emulate them. These peripherals are documented in the SAMD21's datasheet[6], including the different memory-mapped registers and their expected behaviors. Having the datasheet or the reference manual is key when implementing a specific microcontroller or system-on-chip in QEMU.

Therefore, we need to implement the most critical components of this SAMD21 microcontroller to get it working as expected, and especially the following hardware peripherals:

- **System controller**: controls the chip clock sources, brown-out detectors and on-chip voltage regulator .
- **Generic clock controller**: provides nine different clock sources that can be configured to clock peripherals with a specific clock source.
- **Power manager**: controls the reset, clocks generation and sleep modes of the device.
- **I/O pin controller**: controls the physical inputs and outputs of the SAMD21 microcontroller as well as multiplexing.
- **Serial communication controller**: provides a communication interface that supports USART and SPI protocols.

SAMD21 microcontroller provides a lot more hardware peripherals (digital-to-analog converters, timers, USB controller, analog-to-digital converters, peripheral touch interface, etc.) but since we do not need all of them they will be left unimplemented, and thus, will not be detailed in this paper.

3.2.1 Defining a new class for our microcontroller

Let's start implementing this SAMD21 microcontroller by adding in QEMU a dedicated source file specifying a basic component class. We create the `/hw/arm/samd21.c` file and declare a new type for this microcontroller unit (MCU):

```
/* Define a basic SAMD21 component deriving from
   QEMU system bus device class. */
static const TypeInfo samd21_info = {
    .name          = TYPE_SAMD21_MCU,
    .parent        = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(SAMD21State),
    .instance_init = samd21_init,
```

```
    .class_init    = samd21_class_init,
};

/* Declare SAMD21-related types. */
static void samd21_types(void)
{
    type_register_static(&samd21_info);
}

/* Register our SAMD21 component into QEMU. */
type_init(samd21_types)
```

A specific state structure is declared as well, that will represent the state of each instance of this class:

```
struct SAMD21State {
    /*< private >*/
    SysBusDevice parent_obj;

    /*< public >*/
    ARMv7MState cpu;

    /* Memory regions. */
    MemoryRegion iomem;
    MemoryRegion sram;
    MemoryRegion flash;

    /* Properties. */
    uint32_t sram_size;
    uint32_t flash_size;

    /* Board memory region and container. */
    MemoryRegion *board_memory;
    MemoryRegion container;

    /* Main CPU clock. */
    Clock *sysclk;
};
```

As seen in 2.1 this structure contains a first member that corresponds to the component base class, in this case a *QEMU System Bus device*, represented here with the `SysBusDevice parent_obj` member. This structure will be allocated for each instance and keep the associated state in memory.

The `samd21_init` function will be in charge of initializing each instance

of our new MCU class, while the `samd21_class_init` function will be called on class initialization:

```
/* SAMD21 component properties. */
static Property samd21_properties[] = {
    DEFINE_PROP_LINK("memory", SAMD21State, board_memory, TYPE_MEMORY_REGION,
    MemoryRegion *),
    DEFINE_PROP_UINT32("sram-size", SAMD21State, sram_size, SAMD21_X18_SRAM_SIZE),
    DEFINE_PROP_UINT32("flash-size", SAMD21State, flash_size,
    SAMD21_X18_FLASH_SIZE),
    DEFINE_PROP_END_OF_LIST(),
};

static void samd21_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);

    dc->realize = samd21_realize;
    device_class_set_props(dc, samd21_properties);
}
```

This function set the generic properties associated with our new class that can be set by the code creating a new instance of our class using some dedicated QEMU primitives we will cover later in this paper. These properties are mapped to the corresponding fields in our microcontroller state structure `SAMD21State`.

The `samd21_class_init` function also sets the *realize* callback that will be called once an instance of our class has been properly configured. This callback is important as it is in charge of initializing all the internal components including our CPU, memory regions and hardware peripherals when required.

3.2.2 Adding the correct ARM CPU

The `samd21_init` function is used to initialize each instance of our class by configuring the underlying CPU and its properties and setting up the associated clock:

```
static void samd21_init(Object *obj)
{
    SAMD21State *s = SAMD21_MCU(obj);

    /* Declare our container memory region. */
    memory_region_init(&s->container, obj, "samd21-container", UINT64_MAX);
```

```
    /* Initialize an ARMV7M CPU (used in Cortex-M0+ arch) */
    object_initialize_child(OBJECT(s), "armv6m", &s->cpu, TYPE_ARMV7M);
    qdev_prop_set_string(DEVICE(&s->cpu), "cpu-type",
    ARM_CPU_TYPE_NAME("cortex-m0"));
    qdev_prop_set_uint32(DEVICE(&s->cpu), "num-irq", 32);

    /* Initialize our system clock. */
    s->sysclk = qdev_init_clock_in(DEVICE(s), "sysclk", NULL, NULL, 0);
}
```

3.2.3 Defining the microcontroller memory

The microcontroller memory is initialized when a class instance is *realized*, meaning that when our `samd21_realize` callback function is called, it must configure everything. For now, since no hardware peripheral has been implemented, we simply configure the system Flash and SRAM memory regions:

```
static void samd21_realize(DeviceState *dev_mcu, Error **errp)
{
    SAMD21State *s = SAMD21_MCU(dev_mcu);
    Error *err = NULL;

    if (!s->board_memory) {
        error_setg(errp, "memory property was not set");
        return;
    }

    /*
    * HCLK on this SoC is fixed, so we set up sysclk ourselves and
    * the board shouldn't connect it.
    */
    if (clock_has_source(s->sysclk)) {
        error_setg(errp, "sysclk clock must not be wired up by the board code");
        return;
    }

    /* This clock doesn't need migration because it is fixed-frequency */
    clock_set_hz(s->sysclk, HCLK_FRQ);
    qdev_connect_clock_in(DEVICE(&s->cpu), "cpuclk", s->sysclk);

    /* Link container memory to system memory. */
    object_property_set_link(OBJECT(&s->cpu), "memory", OBJECT(&s->container),
    &error_abort);
    if (!sysbus_realize(SYS_BUS_DEVICE(&s->cpu), errp)) {
```

```
        return;
    }

    memory_region_add_subregion_overlap(&s->container, 0, s->board_memory, -1);

    /* Initialize SRAM. */
    memory_region_init_ram(&s->sram, OBJECT(s), "samd21.sram", s->sram_size,
    &err);
    if (err) {
        error_propagate(errp, err);
        return;
    }
    memory_region_add_subregion(&s->container, SAMD21_SRAM_BASE, &s->sram);

    /* Initialize our Flash memory. */
    memory_region_init_rom(&s->flash, OBJECT(s), "samd21.flash",
                           SAMD21_X18_FLASH_SIZE, &err);
    if (err) {
        error_propagate(errp, err);
        return;
    }
    memory_region_add_subregion_overlap(&s->container,
    SAMD21_FLASH_BASE, &s->flash, -1);

    /* Other hardware peripherals are still unimplemented. */
    create_unimplemented_device("samd21_mcu.io", SAMD21_IOMEM_BASE,
    SAMD21_IOMEM_SIZE);
}
```

Additional hardware peripheral implementations will also be initialized and mapped in memory in this function.

3.2.4 System controller (SYSCTRL)

The SAMD21 system controller exposes multiple memory-mapped registers allowing the application code to configure the device. We need to mimic the behavior of this hardware peripheral to get the code working as expected.

In fact, we don't need to implement the complete behavior as most of the startup code included in most firmwares only read registers and don't really *interact* with the peripheral. If we configure the registers with the correct values in memory and allow the application to read and write into them, everything will work as expected.

First, we declare the system controller registers offsets in a header file, based on the datasheet:

```
/* Define SYSCTRL registers offsets. */
REG32(SYSCTRL_INTENCLR, 0x00)
REG32(SYSCTRL_INTENSET, 0x04)
REG32(SYSCTRL_INTFLAG, 0x08)
REG32(SYSCTRL_PCLKSR, 0x0C)
REG32(SYSCTRL_XOSC, 0x10)
// ... skipped code ...
REG32(SYSCTRL_DPLLRATIO, 0x48)
REG32(SYSCTRL_DPLLCTRLB, 0x4C)
REG32(SYSCTRL_DPLLSTATUS, 0x50)
```

We add an array to hold the values of the system controller peripheral in our `SAMD21State` structure:

```
/* SYSCTRL registers. */
uint32_t sysctrl_regs[0x15];
```

Then we initialize them in `samd21_realize` with the defaults values except for the ready bits we want to force:

```
/* Initialize our system controller (SYSCTRL) MMIO. */
memset(&s->sysctrl_regs, 0, sizeof(s->sysctrl_regs));

// Reset registers to their initial value
s->sysctrl_regs[R_SYSCTRL_XOSC] = 0x0080;
s->sysctrl_regs[R_SYSCTRL_XOSC32K] = 0x0080;
s->sysctrl_regs[R_SYSCTRL_OSC32K] = 0x003F0080;
s->sysctrl_regs[R_SYSCTRL_OSC8M] = 0x00000382;
s->sysctrl_regs[R_SYSCTRL_DFLLCTRL] = 0x0080;
s->sysctrl_regs[R_SYSCTRL_VREG] = 0x0002;
s->sysctrl_regs[R_SYSCTRL_DPLLCTRLA] = 0x80;

/* Mark all clocks as enabled and ready by default
   (initialization bypass) */

/* EN32K=1 and ENABLE=1 */
s->sysctrl_regs[R_SYSCTRL_XOSC32K] |= 0x06;

/* DFLLRDY=1, OSC32KRDY=1 and OSC8MRDY=1 */
s->sysctrl_regs[R_SYSCTRL_PCLKSR] |= 0x1C;
```

Eventually, we tell QEMU to call some special callbacks on every read or write operation performed on the SAMD21 system controller's memory-mapped registers:

```
/*****************************************************************************
* SAMD21 System Control (SYSCTRL)
*****************************************************************************/

static uint64_t sysctrl_read(void *opaque, hwaddr addr, unsigned int size)
{
    SAMD21State *s = SAMD21_MCU(opaque);

    /* Return register value. */
    return s->sysctrl_regs[addr/4];
}

static void sysctrl_write(void *opaque, hwaddr addr, uint64_t data,
unsigned int size)
{
    SAMD21State *s = SAMD21_MCU(opaque);
    s->sysctrl_regs[addr/4] = (data & 0xffffffff);
}

/* Memory operation handlers for SYSCTRL. */
static const MemoryRegionOps sysctrl_ops = {
    .read = sysctrl_read,    /* read operation handler. */
    .write = sysctrl_write,    /* write operation handler. */
    .endianness = DEVICE_NATIVE_ENDIAN,
    .impl.min_access_size = 4,    /* Min access size is 4 bytes */
    .impl.max_access_size = 4   /* Max access size is 4 bytes */
};

static void samd21_realize(DeviceState *dev_mcu, Error **errp)
{
    SAMD21State *s = SAMD21_MCU(dev_mcu);
    Error *err = NULL;

    // ... skipped code ...

    /*
     Tell QEMU to call our handlers if any read/write is requested
     on our SYSCTRL peripheral registers.
    */
    memory_region_init_io(&s->sys, OBJECT(dev_mcu), &sysctrl_ops,
                          (void *)s, "samd21.sysctrl",
                          SAMD21_SYSCTRL_PERIPH_SIZE);

    /*
```

```
     Add our SYSCTRL memory region into our container.
    */
    memory_region_add_subregion_overlap(&s->container,
    SAMD21_SYSCTRL_BASE, &s->sys, -1);

}
```

3.2.5 Generic clock controller (GCLK)

Adding the SAMD21 generic clock controller is very similar to what we did with the system controller, as most of the application bootstrap code only performs only read operations. The implementation follows the same scheme: we first define the registers offsets, create a structure in our microcontroller state structure to store the registers in memory and tell QEMU to map every read/write operation to this structure.

This implementation has not been detailed here for clarity purpose since it follows the same pattern as above, but will be included in the final version of this paper.

3.2.6 Power manager

SAMD21's power manager initialization is handled very easily by the official application bootstrap code, as shown in the following disassembled code extracted from a test firmware we built with Microchip's *MPLabX* integrated development environment:

```
                    *************************************************************
                    *                          FUNCTION                         *
                    *************************************************************
                    undefined _pm_init()
                       assume LRset = 0x0
                       assume TMode = 0x1
         undefined     r0:1           <RETURN>
                    _pm_init+1                                  XREF[2,1]:   Entry Point(*),
                    _pm_init                                                 _init_chip:00000262(c),
                                                                             _init_chip:00000260(*)
    000002dc 06 4b      ldr         r3,[DAT_000002f8]                           = 40000400h
    000002de 1a 7a      ldrb        r2,[r3,#offset DAT_40000408]
    000002e0 d2 b2      uxtb        r2,r2
    000002e2 1a 72      strb        r2,[r3,#offset DAT_40000408]
    000002e4 5a 7a      ldrb        r2,[r3,#offset DAT_40000409]
    000002e6 d2 b2      uxtb        r2,r2
    000002e8 5a 72      strb        r2,[r3,#offset DAT_40000409]
    000002ea 9a 7a      ldrb        r2,[r3,#offset DAT_4000040a]
    000002ec d2 b2      uxtb        r2,r2
    000002ee 9a 72      strb        r2,[r3,#offset DAT_4000040a]
    000002f0 da 7a      ldrb        r2,[r3,#offset DAT_4000040b]
    000002f2 d2 b2      uxtb        r2,r2
    000002f4 da 72      strb        r2,[r3,#offset DAT_4000040b]
    000002f6 70 47      bx          lr
```

Figure 3: *Microchip MPLabX* bootstrap code for SAMD21 Power Manager controller

Defining a memory region at the expected base address initialized with zeroes is strictly enough to get the application code started. This is done similarly to the clock controller or the system controller, see 3.2.4.

This implementation has not been detailed here for clarity purpose since it follows the same pattern as above, but will be included in the final version of this paper.

3.2.7 SERCOM peripheral

The SAMD21 *SERCOM* peripheral provides an interface for different *ser*ial *com*munication protocols including the *Universal Synchronous/Asynchronous Receiver Transmitter* protocol (USART) and the *Serial Peripheral Interface* (SPI) protocol. The same hardware peripheral can be configured to interact with external components, be it some SPI Flash memory chip or a computer. Microchip's SAMD21 has 6 SERCOM hardware peripherals (*SERCOM0* to *SERCOM5*) defined in its specification but not all chip variants provide the corresponding physical inputs and outputs.

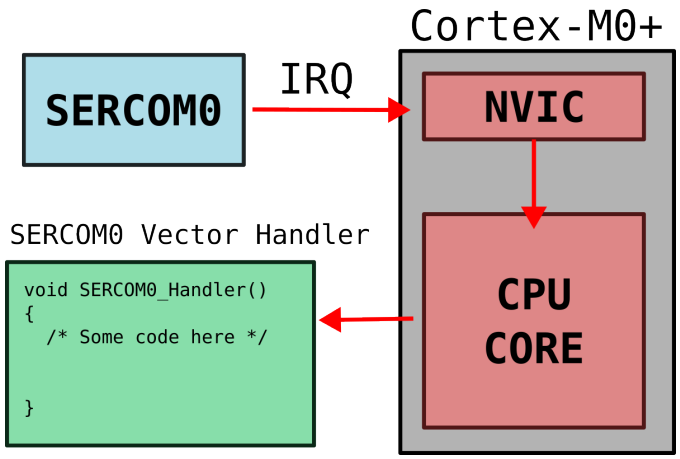

Figure 4: From *SERCOM0* IRQ to handler execution

SAMD21's SERCOM peripheral has one interrupt line (IRQ) connected to the main CPU Nested Vector Interrupt Controller (NVIC) that is triggered when specific events occur and conditions are met. This allows the associated vector handler to be called in order to handle a specific event that happened, such as some data received or any error that may happen (as shown in figure 4). In QEMU, this IRQ is part of the hardware peripheral model and can be connected to the corresponding NVIC interrupt line depending on the peripheral number. The figure 5 taken from the datasheet shows the interrupted lines associated with each *SERCOM* peripheral.

EVSYS – Event System	8
SERCOM0 – Serial Communication Interface 0	9
SERCOM1 – Serial Communication Interface 1	10
SERCOM2 – Serial Communication Interface 2	11
SERCOM3 – Serial Communication Interface 3	12
SERCOM4 – Serial Communication Interface 4	13
SERCOM5 – Serial Communication Interface 5	14
TCC0 – Timer Counter for Control 0	15

Figure 5: *SERCOM* peripherals NVIC interrupt lines as specified in the datasheet

As other hardware peripherals, the SAMD21 SERCOM hardware peripheral has its own set of memory-mapped registers we need to emulate. Again, we follow as strictly as possible the datasheet to implement the correct behavior, but we also need this time to interact with QEMU as we want the SERCOM0 USART interface to be available to the user in a terminal. Instead of outputting bytes to some GPIOs, we will send directly the bytes sent over one USART interface (in our case, *SERCOM0*) to an emulated serial interface handled by QEMU. If this serial interface is set to be the standard output by the user, then the produced text output will be displayed in the terminal. In a similar manner, we want any input provided by the user in the terminal (using the standard input) to be fed into our *SERCOM0* hardware peripheral in order for the firmware to catch this data and process it as if it was sent over a real USART interface.

When a byte is sent through the *SERCOM* peripheral, we simply tell QEMU that the associated *character device* has received one byte of data:

```
static gboolean uart_transmit(void *do_not_use, GIOCondition cond,
                              void *opaque)
{
    SAMD21SERCOMState *s = SAMD21_SERCOM(opaque);
    int r;

    /* Extract the byte written into the DATA register. */
    uint8_t c = s->reg16[R_SERCOM_DATA];

    /* Reset DRE bit of INTFLAG (no new byte can be sent). */
    s->reg8[R_SERCOM_INTFLAG] &= (~(1 << R_SERCOM_INTFLAG_DRE_SHIFT) |
        (1 << R_SERCOM_INTFLAG_TXC_SHIFT));

    s->watch_tag = 0;
    /* Send this byte to QEMU associated character device */
    r = qemu_chr_fe_write(&s->chr, &c, 1);
```

```
    if (r <= 0) {
        /* If an error occured, try to retransmit later or drop the byte. */
        s->watch_tag = qemu_chr_fe_add_watch(&s->chr, G_IO_OUT | G_IO_HUP,
        uart_transmit, s);
        if (!s->watch_tag) {
            /* The hardware has no transmit error reporting,
            * so silently drop the byte
            */
            goto buffer_drained;
        }
        return G_SOURCE_REMOVE;
    }

    buffer_drained:
    /* Set interrupt flags to notify this byte has been sent (DRE) and
       transmission is complete (TXC). */
    s->reg8[R_SERCOM_INTFLAG] |= (1 << R_SERCOM_INTFLAG_DRE_SHIFT);
    s->reg8[R_SERCOM_INTFLAG] |= (1 << R_SERCOM_INTFLAG_TXC_SHIFT);
    s->reg16[R_SERCOM_DATA] = 0;
    s->pending_tx_byte = false;
    return G_SOURCE_REMOVE;
}
```

When data is received on the configured serial interface, we interact with the peripheral registers and notify the hardware that an event occured (if corresponding interrupts have been enabled by software):

```
static void uart_receive(void *opaque, const uint8_t *buf, int size)
{
    SAMD21SERCOMState *s = SAMD21_SERCOM(opaque);
    int i;

    /* Sanity checks (exit if no byte received or RX FIFO is full). */
    if (size == 0 || s->rx_fifo_len >= UART_FIFO_LENGTH) {
        return;
    }

    /* Load received bytes into RX FIFO. */
    for (i = 0; i < size; i++) {
        uint32_t pos = (s->rx_fifo_pos + s->rx_fifo_len) % UART_FIFO_LENGTH;
        s->rx_fifo[pos] = buf[i];
        s->rx_fifo_len++;
    }
```

```
    /* Set RXC bit (receive complete) in INTFLAG register. */
    s->reg8[R_SERCOM_INTFLAG] |= (1 << R_SERCOM_INTFLAG_RXC_SHIFT);

    /* Trigger IRQ if required. */
    samd21_update_irq(s);
}
```

The corresponding IRQ is triggered if at least one event has been enabled by software:

```
static void samd21_update_irq(SAMD21SERCOMState *s)
{
    uint8_t intflag = s->reg8[R_SERCOM_INTFLAG];
    uint8_t intenset = s->interrupts;
    bool irq = false;

    /* DRE */
    irq |= (intflag &&
    (intflag & R_SERCOM_INTFLAG_DRE_MASK) &&
    (intenset & R_SERCOM_INTFLAG_DRE_MASK));

    /* TXC */
    irq |= (intflag &&
    (intflag & R_SERCOM_INTFLAG_TXC_MASK) &&
    (intenset & R_SERCOM_INTFLAG_TXC_MASK));

    /* RXC */
    irq |= (intflag &&
    (intflag & R_SERCOM_INTFLAG_RXC_MASK) &&
    (intenset & R_SERCOM_INTFLAG_RXC_MASK));

    /* other conditions here ... */

    /* Set SERCOM IRQ line level to 1 or 0. */
    qemu_set_irq(s->irq, irq);
}
```

When `qemu_set_irq()` is called with a level of 1, the corresponding NVIC interrupt line is triggered and this cause the associated vector handler to be executed next. This emulates the way interrupts are triggered on the real hardware and allows the handler code to query this peripheral, retrieve the received byte and process it immediately, interrupting whatever the code was doing. Then the execution resumes to the place it has been interrupted, and the CPU goes on with the next instructions.

Last but not least, we add the corresponding initialization code in the SAMD21 instance initialization callback `samd21_realize()`:

```
/* Realize SERCOM0. */
if (!sysbus_realize(SYS_BUS_DEVICE(&s->sercom), errp)) {
    return;
}

/* Map SERCOM0 to its base address into our container memory. */
mr = sysbus_mmio_get_region(SYS_BUS_DEVICE(&s->sercom), 0);
memory_region_add_subregion_overlap(&s->container, SAMD21_SERCOM0_BASE, mr, 0);

/* Connect SERCOM0's IRQ to the CPU NVIC line #9,
   as defined in the SAMD21 spec. */
sysbus_connect_irq(SYS_BUS_DEVICE(&s->sercom), 0,
                   qdev_get_gpio_in(DEVICE(&s->cpu), 9));
```

This section will also cover in details how we implemented the SAMD21 SERCOM SPI interface and integrated it with a QEMU *Synchronous Serial Interface* bus. This is required to make the SAMD21 communicate with the on-board SPI Flash chip with SERCOM1.

3.3 Defining our board in QEMU

Once the main MCU and its required hardware peripherals have been implemented, defining a board using this MCU and other peripherals is pretty straightforward. It looks like a Lego game as we need to create objects of different classes and interconnect them altogether.

First, we need to declare our board as a new machine, using the QEMU Object Model:

```
static const TypeInfo lilboard_info = {
    .name = TYPE_LILBOARD_MACHINE,
    .parent = TYPE_MACHINE,
    .instance_size = sizeof(LilboardMachineState),
    .class_init = lilboard_machine_class_init,
};

static void lilboard_machine_init(void)
{
    type_register_static(&lilboard_info);
}
```

```
type_init(lilboard_machine_init);
```

As for all other QOM classes, we also defined a specific state structure *LilboardMachineState* as follows:

```
struct LilboardMachineState {
    /* Parent machine state. */
    MachineState parent;

    /* MCU state. */
    SAMD21State samd21;
};
```

3.4 Setting up the MCU, memory and external peripherals

The most interesting code resides in the `lilboard_init` function that creates the main SAMD21 MCU, configures the character device associated to the *SERCOM0* hardware peripheral, initializes the board memory and loads the provided firmware into it:

```
static void lilboard_init(MachineState *machine)
{
    LilboardMachineState *s = LILBOARD_MACHINE(machine);
    MemoryRegion *system_memory = get_system_memory();

    /* Initialize our MCU. */
    object_initialize_child(OBJECT(machine), "samd21", &s->samd21,
    TYPE_SAMD21_MCU);

    /* Set serial device. */
    qdev_prop_set_chr(DEVICE(&s->samd21), "serial0", serial_hd(0));

    /* Memory initialization. */
    object_property_set_link(OBJECT(&s->samd21), "memory",
    OBJECT(system_memory), &error_fatal);

    /* Realize our MCU. */
    sysbus_realize(SYS_BUS_DEVICE(&s->samd21), &error_fatal);

    /* Load kernel (firmware) into memory. */
    armv7m_load_kernel(ARM_CPU(first_cpu), machine->kernel_filename,
    0, s->samd21.flash_size);
}
```

The external SPI flash will be created and connected in this function, using the same hardware GPIO as the original board for the flash chip select line.

4 Debugging our custom board

This section will cover how the previously defined emulated board can be used for debugging and vulnerability research.

4.1 Running a firmware on our emulated board

Following QEMU documentation, running a firmware dumped from a real SAMD21 MCU is quite easy, we simply need to tell QEMU to use our new *qb-lilboard* machine with a serial device bound to the standard input and output, and to use a sample firmware provided with the `kernel` option:

```
$ ./qemu-system-arm -M qb-lilboard -serial stdio -kernel ./firmware.bin
=== SAMD21 SSTIC 2024 Crackme ===

Enter password:
```

We can interact with this firmware by typing in the terminal as the standard input and output are automatically linked to the device *SERCOM0* hardware interface:

```
./qemu-system-arm -M qb-lilboa -kernel /tmp/test.bin
=== SAMD21 SSTIC 2024 Crackme ===

Enter password: ThisIsAtest
Nope :(
Enter password:
```

4.2 Debugging a firmware with GDB

QEMU can be started with a GDB server attached using the `-s` option that will open a GDB server on port 1234 allowing to debug the target CPU and memory. It can be combined with the `-S` option that will launch the emulated board in a paused state:

```
$ ./qemu-system-arm -M qb-lilboard -serial stdio -kernel ./firmware.bin -s -S
```

And we can then use `gdb-multiarch` to debug the emulated board microcontroller:

```
(gdb) set arch arm
The target architecture is set to "arm".
(gdb) target remote :1234
Remote debugging using :1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000280 in ?? ()
(gdb) x/20i $pc
=> 0x280:        push        {r4, r5, r6, lr}
0x282:        ldr        r1, [pc, #60]        @ (0x2c0)
0x284:        ldr        r4, [pc, #60]        @ (0x2c4)
0x286:        cmp        r1, r4
0x288:        bne.n        0x294
0x28a:        bl        0x390
0x28e:        bl        0xebc
0x292:        b.n        0x292
0x294:        ldr        r5, [pc, #48]        @ (0x2c8)
0x296:        movs        r3, #0
0x298:        cmp        r5, r1
0x29a:        beq.n        0x28a
0x29c:        adds        r2, r1, r3
0x29e:        adds        r0, r5, r3
0x2a0:        cmp        r2, r4
0x2a2:        bcc.n        0x2b0
0x2a4:        ldr        r3, [pc, #36]        @ (0x2cc)
0x2a6:        cmp        r2, r0
0x2a8:        beq.n        0x28a
0x2aa:        ldr        r1, [pc, #36]        @ (0x2d0)
(gdb) c
Continuing.
```

5 Discussion

QEMU provides a lot more features that may open a world of possibilities to the security researcher, some of them being used by tools like Avatar2[1] for instance, like UNIX or UDP sockets used to allow QEMU to communicate with external tools. We can imagine some interesting possibilities regarding radio transceivers emulation, like offering the possibility to send raw packets normally sent over the air by some hardware peripheral to a UNIX socket and receiving raw packets from the same socket, allowing any external tool to fuzz RF packets and monitor the firmware through GDB.

Some recent research work like GDBFuzz[5] may also been used to perform a greybox fuzzing of a firmware with high performances on a custom

board emulated by QEMU.

6 Conclusion

At first sight, QEMU may look like a huge monster piece of code that would be difficult to apprehend and modify, especially its internals. But QEMU is in fact a clever software with well-thought mechanisms that abstracts most of the hard work and provide the developer with some basic tools that are enough for most of the embedded devices he/she may want to emulate. The QEMU Object Model plays an important role as it allows modularity and reusability: one can add an SPI Flash device to a custom board in a few lines of C or simply assemble a custom board from already existing pieces that can be simply put together and interconnected.

We demonstrated in this paper that a very small subsets of API functions are required to add support for a new microcontroller, its specific hardware peripherals, and a new custom board that supports debugging. We also provide a complete documented example code based on the latest version of QEMU to date (8.2.0) to illustrate the implementation of this custom board we designed for training and vulnerability research.

References

[1] Avatar2. https://github.com/avatartwo/avatar2.

[2] Damien Cauquil. *Meet Piotr, a firmware emulation tool for trainers and researchers*. 2021. URL: https://archives.pass-the-salt.org/Pass%20the%20SALT/2021/slides/PTS2021-Talk-16-piotr.pdf.

[3] Google. *Welcome to the Android Emulator*. https://android.googlesource.com/platform/external/qemu/+/2db80f7c1921a6f5d48b998378e3792e16c968a4/README.md.

[4] Stéphane Duverger (Airbus Security Lab). *QEMU internals*. 2021. URL: https://airbus-seclab.github.io/qemu_blog/.

[5] Max Eisele, Daniel Ebert, Christopher Huth and Andreas Zeller. *Fuzzing Embedded Systems Using Debug Interfaces*. 2023. URL: https://publications.cispa.saarland/3950/1/issta23-gdbfuzz.pdf.

[6] Microchip. https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.pdf.

[7] The QEMU Project. *Internal QEMU APIs*. URL: https://www.qemu.org/docs/master/devel/index-api.html.

[8] The Unicorn Engine Project. https://www.unicorn-engine.org/.

[9] Saumil Shah. *EMUX (formerly ARMX) Firmware Emulation Framework*. URL: https://github.com/therealsaumil/emux.

[10] Kota Shima. *High-Speed Simulator for R-Car S4 - Renesas QEMU Environment*. https://www.renesas.com/us/en/blogs/r-car-s4-renesas-qemu-environment.

[11] Panda Team. *Panda*. https://panda.re/.