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

Damien «virtualabs» Cauquil
dcauquil@quarkslab.com

Quarkslab

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*[12] or the *Unicorn Engine*[9] 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[11], or even Unicorn Engine[9] 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[10]).

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 and also in the official documentation[7].

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 is available on Github [1] with test firmware files and a small guide on how to build Qemu and use this new emulated board.

2 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 allow 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)*.

2.1 QEMU devices lifecycle

In the QEMU Object Model, devices follow a specific lifecycle: they are first created, then initialized, *realized* and optionnally *unrealized*.

The creation step is the most straightforward one as it consists in allocating in memory a structure corresponding to the device class instance. Once allocated, the device is initialized through a call to its *initialize()* callback function. If the corresponding device class has not been already initialized at this point, it is initialized first and then the device gets initialized. This is required as the device callbacks may reference its parent class members.

Once initialized, the device's properties can be set with the help of various *qdev_prop_set*()* functions to assign values to some properties or link them to other existing objects such as memory regions. Setting the object properties is not enough as it will only set some members of its device state structure and will have no real effect until the device's *realize()* callback is called.

Device *realization* consists in setting up the real device based on its properties and its initial state, thus connecting everything and configuring the device

[1] `https://github.com/quarkslab/sstic-tame-the-qemu`

```
struct ADS7846State {
  SSIPeripheral ssidev;
  qemu_irq interrupt;

  int input[8];
  int pressure;
  int noise;

  int cycle;
  int output;
};
```

Listing 1. ADS7846 class definition

for its use in the emulated machine. This step may fail, this is why the *realize()* callback has a dedicated `Error` pointer to report any error that might happen during the device *realization*. Some devices may also need to provide an *unrealize()* callback if they can be unplugged from the machine after creation.

QEMU's official documentation details the QEMU Object Model[8] and provides some example code for new device class implementation including some very specific usages that are not covered in this document.

2.2 Definition of object classes in QEMU

From a technical perspective, those classes are basically defined as C structures, as shown in listing 1. In this example the `SSIPeripheral` structure defines the base class with its members (a serial synchronous interface peripheral), and listing 2 shows how this structure is defined in QEMU. And of course this `SSIPeripheral` structure inherits from the base device class `DeviceClass`, defined with its own structure as shown in listing 3.

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 at 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, initial-

```
struct SSIPeripheral {
  DeviceState parent_obj;

  /* cache the class */
  SSIPeripheralClass *spc;

  /* Chip select state */
  bool cs;

  /* Chip select index */
  uint8_t cs_index;
};
```

Listing 2. SSIPeripheral structure definition

```
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;
```

Listing 3. QEMU DeviceClass structure definition

```
/**
* struct ObjectClass:
*
* The base for all classes.  The only thing that #ObjectClass
* contains is an integer type handle.
*/
struct ObjectClass
{
  /* private: */
  Type type;
  GSList *interfaces;

  const char *object_cast_cache[OBJECT_CLASS_CAST_CACHE];
  const char *class_cast_cache[OBJECT_CLASS_CAST_CACHE];

  ObjectUnparent *unparent;

  GHashTable *properties;
};
```

Listing 4. QEMU ObjectClass type definition

ize 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 5.2).

Using nested structures and specific members for each of them allows QEMU to define a *type hierarchy*, each type being by a structure placed at its beginning that inherits from a base type (in this case, `ObjectClass`) completed with a set of other members defined by each derived type successively. Fig 1 shows the type hierarchy of the *ADS7846* peripheral structure. The resulting structure once unfolded is presented in Fig 2. Using these nested structures, QEMU can easily retrieve the different members corresponding to the related classes as they are stored at specific offsets in memory.

Every declared class inherits from `ObjectClass` (defined in `include/qom/object.h`), a structure that stores critical information regarding an object but also a set of properties that this object exposes stored in its `properties` member (listing 4).

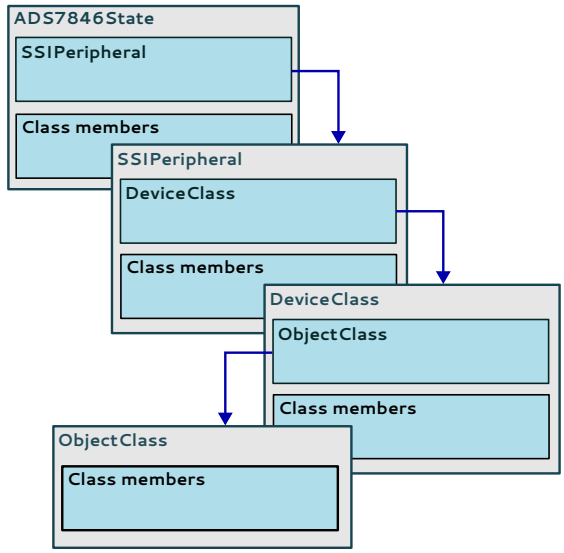

Fig. 1. QEMU type hierarchy example

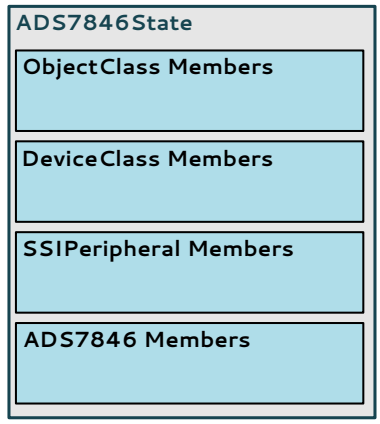

Fig. 2. Resulting structure for ADS7846 peripheral state

2.3 Communication between objects

Creating objects is one thing but allowing them to interact with other objects is another. For instance, we may have a *CPU* defined with a *Nested Vector Interrupt Controller (NVIC)* that manages every possible incoming interrupt request or *IRQ*, but this *NVIC* needs to be connected to some other objects that need to trigger an *IRQ*. This is why QEMU provide support for *GPIO*s and *IRQ*s.

Basically, a *GPIO* is an output line that can hold a value (usually 0 or 1) and can be connected to an *IRQ* of another object. By doing so, each time this *GPIO* value is updated it may trigger a callback associated with the *IRQ* it is connected to and let an object trigger a specific behaviour of another object. Fig 3 shows how multiple peripherals of a machine can be connected to its *NVIC* to trigger a call to the corresponding interrupt handlers through dedicated *IRQ*s. Section 8 below in this document demonstrates a possible use of this mechanism to handle events triggered by the user such as characters sent to a serial port by QEMU.

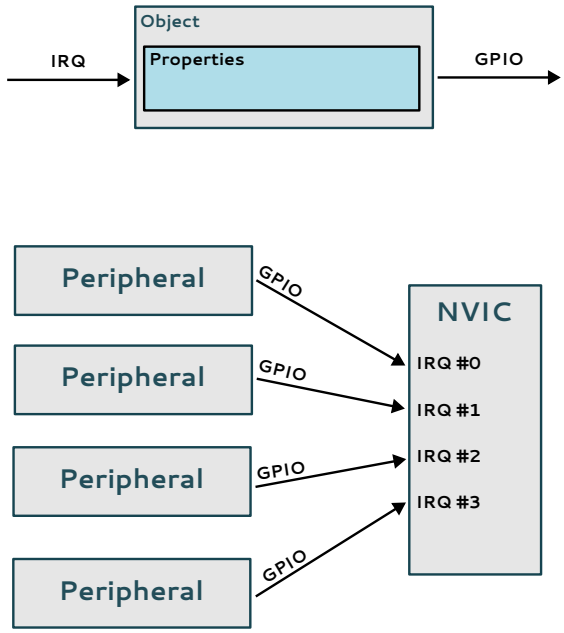

Fig. 3. Example of peripherals connected to a NVIC

2.4 Assembling components like Lego bricks

To create a new custom board we will rely on QEMU basic building blocks: device classes. Since any device exposes its own properties, can be added to

a machine and connected to other devices through *GPIO*s and *IRQ*s, we can elaborate a new emulated machine in which we may create different devices and plug them together to mimic a real machine, therefore providing all the required devices to get some firmware correctly emulated. Just like assembling Lego bricks to build a machine.

QEMU provides a lot of different building bricks:

— Various *CPU*s, *MCU*s and *SoC*s, usually implemented in `hw/_ARCH_/` depending on the target architecture

— External storage devices such as SPI or I^2C Flash memory or NAND memory devices that can be dynamically created and connected to a *SoC* or a *MCU*

— Various UART controllers and devices to support one or more serial interfaces

2.5 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.

/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.

/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.

/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 Methodology

Adding a custom board in QEMU can sometimes be difficult because even if QEMU already supports a lot of different CPU, hardware peripherals and devices, it may lacks the one you need to get your board correctly emulated. It is then important to do a quick inventory of the components required to emulate a target board and determine those that are not currently supported by QEMU and would require some effort to be successfully emulated. A great knowledge of the hardware that needs to be emulated is required in order to precisely determine which components need to be added to QEMU and the amount of effort required to do so.

We will focus on a tiny board we (Quarkslab) designed and use for hardware reverse-engineering purpose and use it as an example throughout this article to illustrate the different steps required to get such a custom board emulated with 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.

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 Hardware overview

From an hardware perspective, this board relies on an *SAMD21* microcontroller and a *W25Q16JV* SPI Flash chip from *Winbond*. The firmware running on the SAMD21 microcontroller can read and write the content of the Flash chip, but also communicate with the user through a dedicated USART (*Univeral Syncrhonous/Asynchronous Receiver/Transmitter*). Fig 5 summarizes the main components that need to be emulated to run a basic firmware:

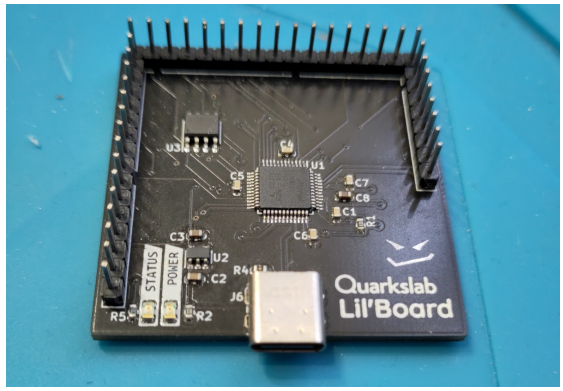

Fig. 4. Quarkslab Lil'board

— **SAMD21**: the microcontroller (*MCU*) this board is based on, that runs a dedicated firmware and interacts with the external SPI Flash.
— **W25Q16**: the SPI Flash chip that contains a defined quantity of information and can be accessed by the *MCU*.
— **CPU**: SAMD21's core CPU, an ARM Cortex-M0+ CPU
— **SRAM**: static RAM used by the CPU to operate
— **FLASH**: SAMD21 internal Flash memory in which the firmware is stored
— **GPIO**: SAMD21's input/output hardware peripheral that controls the *MCU* inputs and outputs (defined as `PORT` in the documentation but renamed here for more readability)
— **SERCOM**: SAMD21's serial communication hardware peripheral that allows the *MCU* to communicate with other external electronics components including SPI Flash chips and USART-compatible devices.

Note that Microchip's SAMD21 *MCU* may also require other core hardware peripherals that are required by the firmware's bootstrap code in charge of initializing the *MCU* but that are not listed here for simplicity. They will be detailed later in this document.

3.3 Determining the missing bricks and their dependencies

In order to emulate a full electronic board, we need to make sure that every single required component is supported. Based on our hardware specification, QEMU already provides the following components:

— A Cortex-M0+ ARM CPU supporting the ARM v6m instruction set, as required by the SAMD21 *MCU*

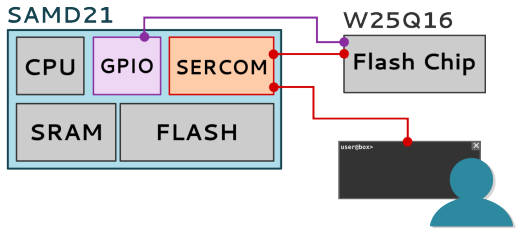

Fig. 5. Overview of our custom board architecture

— A compatible SPI Flash in the form of the *M25P80* SSI slave device (not exactly our *W25Q16JV* chip but it provides others that are fully compatible with the Flash read/write operations)

That means we will need to add the following components to QEMU:

— SAMD21's core peripherals (system controller, power controller and clock controller)
— SAMD21's *GPIO* and *SERCOM* hardware peripherals
— SAMD21 *MCU* using an off-the-shelf ARM Cortex-M0+ CPU with its associated *NVIC* and our newly implemented *GPIO* and *SERCOM* peripherals
— Our custom board that uses our newly implemented SAMD21 MCU and an off-the-shelf SPI Flash chip

3.4 Planning the implementation

One may be tempted to add the required components from bottom to top, starting with SAMD21 core peripherals and its others hardware peripherals and finishing up with the custom board component, but its highly discouraged because of the way memory is managed under QEMU. In QEMU, each memory-mapped object uses its own part of memory that belongs to its parent object and therefore this parent object needs to be defined before in order to test the child object implementation.

We will then start by creating a skeleton for our custom board, with absolutely no CPU or memory at the beginning, and will populate this board with its different components as we add them in QEMU. Doing so will let us define a global memory space for our board and split it among the different components, from top to bottom. It will also ease the development and testing process as we will be able to use our board form the start and debug with QEMU (and GDB) to investigate any error or bug we may encounter.

Based on the various components we previously identified, this is how we plan to implement our custom board:

1. We add our custom board in QEMU by creating a simple skeleton and modifying the build system to include it during compilation
2. We implement our SAMD21 *MCU* object type by creating a simple Cortex-M0+ ARM CPU and by adding memory regions dedicated to SRAM and FLASH
3. We add the capability to load a firmware to our custom board, writing its content into the newly created FLASH memory region
4. We then implement the core controllers as new object types for our SAMD21 *MCU*: a system controller, a clock controller and a power controller that will have their own memory region assigned
5. Once the core controllers implemented, we define a new object type for our *SERCOM* hardware peripheral that supports both USART and SPI protocols, allowing this peripheral to be connected to QEMU's standard serial input/output and offering a dedicated synchronous serial interface (*SSI*)
6. We define a new object type for our *GPIO* hardware peripheral
7. We update our *MCU* implementation in order to add all of the *SERCOM* hardware interfaces and the *GPIO* peripheral
8. We update our custom board implementation to use our latest version of *MCU*, create an external SPI Flash and connect it to our *MCU*

This plan has been designed for our board, depending on its hardware and the missing pieces that need to be implemented. Boards that use standard components such as an already supported SoC like the *nRF51822* for instance or other well-known devices that can be plugged to a QEMU *SSI* bus may be easily implemented and will not require as much development as ours.

4 Step 1: creating a skeleton for our custom board

First thing first, we need to create a basic custom board that provides no CPU and no memory in QEMU, just a board that is known by QEMU and will be made available from QEMU's board list. Since our custom board is based on an ARM CPU, we will consider placing our files in `/hw/arm/` (for source files) and `/include/hw/arm/` (for header files).

As previously explained, our new custom board will be defined as a specific QEMU object type with its own properties, child objects and associated callbacks.

4.1 Board state structure

We create a dedicated file `/hw/arm/qblilboard.c` that will hold the code in charge of our new board. No need of a custom header file, this board will not be used by anything else but the final user through QEMU's CLI. We add the mandatory includes directives as well as a custom structure that will hold our board state (listing 5)

This state structure has a single member declared as a `MachineState` structure and named `parent`, which is mandatory as it is required by the parent type of our board.

4.2 Board type definition

We then add some code to declare our new object type and its associated name (listing 6), declaring a new `TYPE_LILBOARD_MACHINE` macro that will hold our board name. We also add some code to populate a new `TypeInfo` structure describing our new object type and some basic callbacks required to initialize the board (listing 7).

Our new type declaration requires a first callback function to be implemented, *lilboard_machine_class_init()* (listing 8). This function is called by QEMU on type initialization, and will set the machine display name, the type instance initialization callback and the maximum number of CPUs of the board. The instance initialization callback, *lilboard_init()*, will be in charge of the board initialization but is now defined as an empty function (listing 9). This callback function will be called by QEMU when our custom board is instanciated to setup everything before running the firmware.

4.3 Modifying the build system

QEMU uses *Meson* as its main build system, we need to modify one of its configuration file to include our custom board in the build process:

```c
#include "qemu/osdep.h"
#include "qapi/error.h"
#include "hw/qdev-properties.h"
#include "hw/boards.h"
#include "hw/arm/boot.h"
#include "sysemu/sysemu.h"
#include "exec/address-spaces.h"
#include "hw/qdev-properties.h"
#include "qom/object.h"

/**
* Like our SAMD21 SERCOM device class or our MCU device class,
* we need to define a specific device class for our board, as
↪  well
* as a device state structure:
*/

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

Listing 5. Qb Lilboard state structure

```c
#define TYPE_LILBOARD_MACHINE MACHINE_TYPE_NAME("qb-lilboard")
OBJECT_DECLARE_SIMPLE_TYPE(LilboardMachineState,
↪  LILBOARD_MACHINE)
```

Listing 6. Custom board type creation

```
/**
* The following structure describes our machine class type:
*
* - It sets the machine name ("qb-lilboard").
* - It sets the parent class type (a QEMU machine).
* - It sets the instance size (the size of our state structure).
* - It sets the class initialization callback.
*
*/

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

/**
* The following function is in charge of declaring every types related
* to our class.
*
* In fact, it only registers our machine type.
*/

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

/* Tells QEMU to register our module type registration callback. */
type_init(lilboard_machine_init);
```

Listing 7. Custom board type definition

```
static void lilboard_machine_class_init(ObjectClass *oc, void
↪  *data)
{
        MachineClass *mc = MACHINE_CLASS(oc);

        mc->desc = "Qb Lil'board (SAMD21)";
        mc->init = lilboard_init;
        mc->max_cpus = 1;
}
```

Listing 8. Type initialization callback function

```
static void lilboard_init(MachineState *machine)
{
}
```

Listing 9. Type instance initialization callback function

`/hw/arm/meson.build`. We add a line telling *Meson* to include our C file when the `CONFIG_QBLILBOARD` option is set (listing 10).

4.4 Building QEMU

Following QEMU build instructions, we build our modified version of QEMU for ARM targets (listing 11). Once completed, running *qemu-system-arm* with the *–machine help* option displays every supported machine types, including our new custom board !

```
[...]
arm_ss.add(when: 'CONFIG_VEXPRESS', if_true:
↪  files('vexpress.c'))
arm_ss.add(when: 'CONFIG_ZYNQ', if_true: files('xilinx_zynq.c'))
arm_ss.add(when: 'CONFIG_SABRELITE', if_true:
↪  files('sabrelite.c'))
arm_ss.add(when: 'CONFIG_QBLILBOARD', if_true:
↪  files('qblilboard.c'))
```

Listing 10. Meson build script for ARM boards

```
mkdir build && cd build
../configure --target-list=arm-softmmu
make -j4
```

Listing 11. Building QEMU for arm targets

5 Step 2: adding a new SAMD21 MCU in QEMU

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. Fig 6 shows the existing parts (grey background) and those we need to implement (white background).

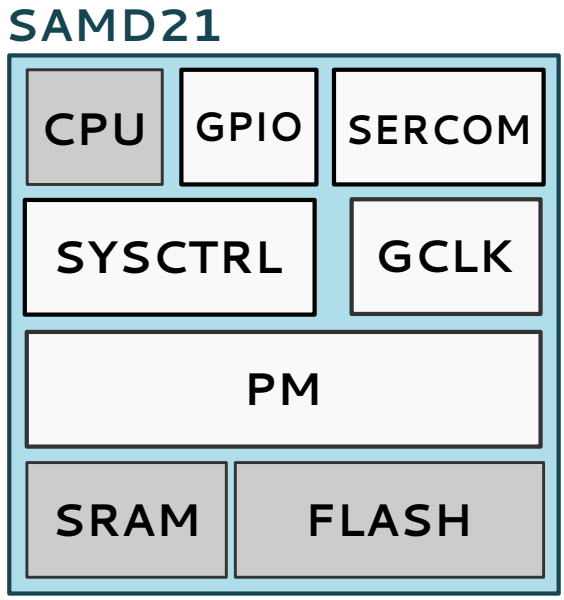

Fig. 6. SAMD21 MCU overview

5.1 Creating a skeleton for our SAMD21 MCU

First, we need to create a dedicated source file for this SAMD21 microcontroller in order to implement a new object type in `/hw/arm/`. Let's create a file named `samd21.c` with the minimum object type declaration code needed (listing 12). We also define a dedicated header file in `/include/hw/arm/samd21_mcu.h` (listing 13).

We add some basic functions required for class and instance initialization, *samd21_class_init()* and *samd21_init()* respectively and we now have a basic SAMD21 MCU object type (listing 14).

5.2 Let's add an ARM Cortex-M0+ CPU

Our code skeleton for this SAMD21 *MCU* can now be improved by creating an ARM Cortex-M0+ CPU during instance initialization. We modify its state structure to include our future ARM CPU state structure that will be initialized when our SAMD21 MCU is initialized (listing 13). Note we use a `ARMv7MState` structure as it also supports ARM v6 instruction set. Note that the `/include/hw/arm/armv7m.h` file is included from our SAMD21 header file.

This new CPU state structure must be initialized each time an instance of our SAMD21 MCU is created, so we do this in the *samd21_init()* function (listing 16). We use QEMU's *object_initialize_child()* function to create a new child object that will be attached to our MCU object of type `TYPE_ARMV7M` and *qdev_prop_set_string()* (line 6) and *qdev_prop_set_uint32()* (line 8) to set the correct CPU type and number of IRQ.

This is basically how we reuse an object defined in QEMU: we initialize a child object by providing a pointer to the corresponding state structure and then we set its properties before *realizing* the object itself. In our case, our child CPU will be added to our system bus and will provide code emulation. The child object *realization* is performed through a call to its *realization* callback that will consider its properties and configure the object instance accordingly.

So we now have a CPU inside our MCU, but we have no firmware to execute and no memory defined where this firmware can be stored. We need to fix this to get some code emulated.

5.3 Defining SAMD21 MCU memory regions

In order for our CPU to work properly, we need to define at least one memory region that will contain some code and that is exposed to our CPU. This is done by adding a new memory region `container` into our SAMD21 state structure (listing 21) and initializing it in *samd21_init()* (listing 20). Once initialized, this memory region can be given to the CPU during our *MCU realization* in order for it to be able to access its virtual memory fron the emulated code (listing 18) thanks to the *object_poperty_set_link()* (line 7).

We also need to define two extra memory regions: one for the SAMD21 static RAM (SRAM) and another one for the SAMD21 embedded Flash. This is

```
#include "qemu/osdep.h"
#include "qapi/error.h"
#include "hw/arm/boot.h"
#include "hw/sysbus.h"
#include "hw/qdev-clock.h"
#include "hw/misc/unimp.h"
#include "qemu/log.h"

/* Include our SAMD21 header files. */
#include "hw/arm/samd21_mcu.h"

/* ... */

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,
};

static void samd21_types(void)
{
        type_register_static(&samd21_info);
}

type_init(samd21_types)
```

Listing 12. SAMD21 skeleton code

```
#ifndef SAMD21_MCU_H
#define SAMD21_MCU_H

#include "hw/sysbus.h"
#include "hw/arm/armv7m.h"
#include "hw/clock.h"
#include "qom/object.h"

#define TYPE_SAMD21_MCU "samd21-mcu"
OBJECT_DECLARE_SIMPLE_TYPE(SAMD21State, SAMD21_MCU)

/* SAMD21 state structure. */
struct SAMD21State {
        /*< private >*/
        SysBusDevice parent_obj;

        /*< public >*/
};

#endif /* SAMD21_MCU_H */
```

Listing 13. SAMD21 MCU header file

```
static void samd21_init(Object *obj)
{
}

static void samd21_realize(DeviceState *dev_mcu, Error **errp)
{
}

static Property samd21_properties[] = {
        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);
}
```

Listing 14. SAMD21 class and instance callbacks

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

        /*< public >*/

        /* We need an ARMv7M CPU object. */
        ARMv7MState cpu;
};
```

Listing 15. ARM CPU state member added to SAMD21State

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

  object_initialize_child(OBJECT(s), "armv7m", &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);
}
```

Listing 16. ARM CPU initialization

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

        /*< public >*/

        /* We need an ARMv7M CPU object. */
        ARMv7MState cpu;

        /* Our main MCU memory container. */
        MemoryRegion container;
}
```

Listing 17. SAMD21 MCU memory container

```
/* Link container memory to system memory. */
object_property_set_link(OBJECT(&s->cpu), "memory",
↪  OBJECT(&s->container),
&error_abort);
```

Listing 18. Setting ou ARM CPU memory region

```
/* Initialize SRAM memory region. */
memory_region_init_ram(&s->sram, OBJECT(s), "samd21.sram",
 ↪  s->sram_size,
&err);
if (err) {
        error_propagate(errp, err);
        return;
}

/* Map the SRAM memory region in our memory container at the
 ↪  correct
base address (provided in the SAMD21 datasheet). */
memory_region_add_subregion(&s->container, SAMD21_SRAM_BASE,
 ↪  &s->sram);
```

Listing 19. SRAM memory region initialization and mapping

done by defining two `MemoryRegion` structures inside our MCU state structure, initializing them in *samd21_init()* and mapping them into our MCU memory space represented by our `container` memory region (listing 19). SRAM and Flash memory regions are mapped using *memory_region_add_subregion()* at their respective memory addresses as specified in the *MCU* datasheet.

The listing 19 uses a member from `SAMD21State` structure named `sram_size`, that has been added to allow our SAMD21 object to have its SRAM size configurable through its properties. We declared `sram_size` as an unsigned 32-bit integer in `SAMD21State` and added a property named `sram-size` to our object type properties, as shown in listing 22. We also added a property named `flash-size` to allow flash size configuration, and a property named `memory` linked to a memory region declared in our state structure as `board_memory`. The latter memory region will be mapped into our container and allow the code creating an instance of our MCU to specify its memory region previously initialized with code and data.

5.4 Defining a clock for our CPU

Our ARM CPU needs a system clock to run properly, therefore we must create a system clock object and assign it to the corresponding CPU property. Creating an input system clock is quite easy, a simple call to *qdev_init_clock_in()* is enough to create such a clock. This function also needs a dedicated `Clock` structure that will hold the clock status, we added one into our `SAMD21State`

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

  /* Initialize the MCU memory container. */
  memory_region_init(&s->container, obj,
    "samd21-container",
    UINT64_MAX);

  object_initialize_child(OBJECT(s), "armv7m", &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);
}
```

Listing 20. Initializing memory region during init

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

  /*< public >*/

  /* We need an ARMv7M CPU object. */
  ARMv7MState cpu;

  /* Our main MCU memory container. */
  MemoryRegion container;
}
```

Listing 21. SAMD21 MCU memory container

```
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(),
};
```

Listing 22. SAMD21 final object properties

```
/*
 * Initialize a clock for this device called "sysclk".
 */
s->sysclk = qdev_init_clock_in(DEVICE(s), "sysclk", NULL, NULL,
↪  0);
```

Listing 23. CPU input clock initialization

structure and initialize this system clock in our *samd21_init()* function (listing 23). We then connect this system clock to our ARM CPU with a call to *qdev_connect_clock_in()* in *samd21_realize()*.

The frequency of this clock is set with a call to *clock_set_hz()*, which is 48MHz based on the *MCU* datasheet.

Now that our CPU has its memory and clock defined, we can *realize* in our *MCU* realization callback *samd21_realize()* (listing 24).

5.5 Using our MCU in our custom board type

The SAMD21 *CPU* now has its clock and memory region defined and is realized when the *MCU* is realized. SRAM and Flash memory regions are also defined as subregions of the *MCU* virtual memory container, this first version of the SAMD21 *MCU* can now be used in our custom board implementation. Note that we only have a CPU, SRAM and Flash memory and no hardware peripheral so we cannot expect it to run smoothly a firmware designed for the SAMD21, but we then can run some code and debug it. Well, once we have an instance of this *MCU* defined in our board and are able to load some code in memory.

```
/* CPU has been correctly configured, we call
   its realize callback and exit if an error
   occurs during this operation. */
if (!sysbus_realize(SYS_BUS_DEVICE(&s->cpu), errp)) {
        return;
}
```

Listing 24. CPU realization

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

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

Listing 25. Adding a SAMD21 MCU state structure in our SAMD21State structure

Since our board needs to create a SAMD21 *MCU* object instance we need to add the corresponding state structure in our `SAMD21State` structure (listing 25). We also update our *lilboard_init()* function to create a SAMD21 *MCU* and realize it (listing 26).

Now our *MCU* will be initialized when our custom board is initialized, some of its properties will be set to get it running as expected and it will be ready to emulate some code. It does not support any hardware peripheral yet, but this will be added later.

5.6 Updating QEMU build system

We also need to update QEMU's build configuration files to add our *MCU* to the compilation targets. We update the `/hw/arm/meson.build` configuration file to include our SAMD21 *MCU* source file (listing 27).

```
/*
 * We retrieve the system memory region allocated to this
 * machine. QEMU allocates by itself the machine' system
 * memory (in fact a structure representing the system
 * memory space, not pre-allocating a huge chunk of memory).
 */
MemoryRegion *system_memory = get_system_memory();

/* Create a SAMD21 MCU. */
object_initialize_child(OBJECT(machine), "samd21", &s->samd21,
TYPE_SAMD21_MCU);

/**
 * After that, we link our MCU "property" to the board system
 * memory, in order for our MCU to be able to access our board
 * memory map.
 */
object_property_set_link(OBJECT(&s->samd21), "memory",
OBJECT(system_memory), &error_fatal);

/**
* When all the properties have been set, we can ask QEMU to
* realize our MCU: it will call the object `realize`
* callback in charge of setting the object state and child
* objects based on the properties we defined.
*/
sysbus_realize(SYS_BUS_DEVICE(&s->samd21), &error_fatal);
```

Listing 26. MCU initialization in our custom board

```
1 arm_ss.add(when: 'CONFIG_NRF51_SOC', if_true:
  ↪  files('nrf51_soc.c'))
2 arm_ss.add(when: 'CONFIG_XEN', if_true: files('xen_arm.c'))
3 arm_ss.add(when: 'CONFIG_SAMD21', if_true: files('samd21.c'))
```

Listing 27. QEMU Meson build configuration file for ARM targets

6 Step 3: loading a firmware into memory

Now that we have our *MCU* added to our board, we can load a firmware file into the board memory. Firmware files are usually provided through QEMU CLI `--kernel` option that defines the machine's `kernel_filename` property of the corresponding state structure given to *lilboard_init()*. That means we will be able to initialize our custom board with a specific firmware like this:

```
$ qemu-system-arm --machine qb-lilboard --kernel my_firmware.hex
```

There is no check by default performed on this kernel file, but we will use a specific function provided by the ARM CPU implementation to load a file into memory, *armv7m_load_kernel()*. This function tries to load the provided file as an ELF file and will fall back to other file formats like *Intel HEX* if an error occurs. Therefore, multiple file formats are supported:

— ELF: compiled ELF file with or without symbols
— Intel HEX: another type of file produced by a compiler for chip programming purpose
— Raw binary: usually what we get when we dump the memory of a *MCU* with a programmer

Firmware loading is performed in *lilboard_init()* and is quite straightforward, we simply need to tell this function to load the provided file at memory address 0x00000000 with maximum size corresponding to the *MCU* flash size for the first CPU defined in our machine.

```
armv7m_load_kernel(ARM_CPU(first_cpu), machine->kernel_filename,
  0, s->samd21.flash_size);
```

7 Step 4: implementing SAMD21 core peripherals

The SAMD21 *MCU* has multiple hardware peripherals including some of them critical for its operation:

— **SYSCTRL**: the main system controller as defined in datasheet section 17
— **GCLK**: clock controller that controls the various clocks and dividers as defined in datasheet section 15
— **PM**: power manager controller as defined in datasheet section 16

These controllers are specific peripherals that control how the microcontroller works and may have critical impact on its behavior. These peripherals must be initialized before using any other peripheral and therefore are usually set up first by a bootstrap code. This bootstrap code is usually automatically included at compilation time and cannot be easily modified by an application developer.

We are going to implement these controllers as part of our SAMD21 *MCU* in `hw/arm/samd21.c`.

7.1 System controller (SYSCTRL)

We don't need to implement the complete behavior of *SYSCTRL* 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, in `include/hw/arm/samd21_mcu.h` (listing 28). We then add an array to hold the values of the system controller peripheral in our `SAMD21State` structure (listing 29) and initialize them in *samd21_realize()* with the defaults values except for the ready bits that we want to force, as the system is supposed to be fully initialized (listing 30).

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 by defining a `MemoryRegionOps` structure as shown in listing 31. This structure references two callback functions: *sysctrl_read()* that controls any read operation performed on the memory region and *sysctrl_write()* that handles any write operation performed on any MMIO register. Since we don't need to fully emulate the *SYSCTRL* peripheral, we simply have to keep track of the registers value (listing 32). Eventually, we update our *samd21_realize()* function to initialize our SYSCTRL peripheral memory and its associated read/write handlers (listing 33).

```
/* 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)
```

Listing 28. SYSCTRL registers declaration

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

Listing 29. SYSCTRL registers array definition

```
/* 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;
```

Listing 30. SYSCTRL registers initialization

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

Listing 31. SYSCTRL `MemoryRegionOps` definition

```
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);
}
```

Listing 32. SYSCTRL register read/write callback functions

```
1  static void samd21_realize(DeviceState *dev_mcu, Error **errp)
2  {
3    SAMD21State *s = SAMD21_MCU(dev_mcu);
4    Error *err = NULL;
5
6    // ... skipped code ...
7
8    /*
9      Tell QEMU to call our handlers if any read/write is
10     requested on our SYSCTRL peripheral registers.
11   */
12
13   memory_region_init_io(&s->sys, OBJECT(dev_mcu),
14     &sysctrl_ops, (void *)s, "samd21.sysctrl",
15     SAMD21_SYSCTRL_PERIPH_SIZE);
16
17   /*
18    Add our SYSCTRL memory region into our container.
19    */
20   memory_region_add_subregion_overlap(&s->container,
21   SAMD21_SYSCTRL_BASE, &s->sys, -1);
22 }
```

Listing 33. SYSCTRL peripheral initialization

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

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

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

static const MemoryRegionOps clock_ops = {
        .read = clock_read,
        .write = clock_write,
        .endianness = DEVICE_NATIVE_ENDIAN,
        .impl.min_access_size = 4,
        .impl.max_access_size = 4
};
```

Listing 34. GCLK memory read/write handlers

7.2 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.

We define a new `MemoryRegionOps` structure for this peripheral (listing 34), and define the corresponding memory region in our *samd21_init()* function (listing 35).

```
/*
 * Initialize our generic clock controller (GCLK) MMIO.
 *
 * We first initialize our MMIO registers to zero and then
 * create a dedicated MMIO region with specific read/write
 * handlers. Then this region is mapped into our memory
 * container at the expected address (again as defined in the
 * datasheet).
 */

memset(s->gclk_regs, 0, sizeof(s->gclk_regs));
memory_region_init_io(&s->clock, OBJECT(dev_mcu), &clock_ops,
 ↪  (void *)s,
"samd21.clock", SAMD21_GCLK_PERIPH_SIZE);

/* Map our GCLK MMIO registers into our memory container. */
memory_region_add_subregion_overlap(&s->container,
SAMD21_GCLK_BASE, &s->clock, -1);
```

Listing 35. GCLK MMIO registers mapping

```
create_unimplemented_device("samd21_mcu.io", SAMD21_IOMEM_BASE,
SAMD21_IOMEM_SIZE);
```

Listing 36. SAMD21 unimplemented devices support

7.3 Power manager

SAMD21's power manager initialization is handled very easily by the official application bootstrap code, as shown in the disassembled code extracted from a test firmware we built with Microchip's *MPLabX* integrated development environment (fig. 7).

Defining a memory region at the expected base address initialized with zeroes is strictly enough to get the application code started. We do not need any read/write memory handlers to handle this peripheral, so we use it as an excuse to add a wide memory region to handle any unimplemented peripherals (listing 36). We use QEMU *create_unimplemented_device()* function to create a default memory region starting at address `SAMD21_IOMEM_BASE` (0x40000000) with a size of `SAMD21_IOMEM_SIZE` (0x02010000).

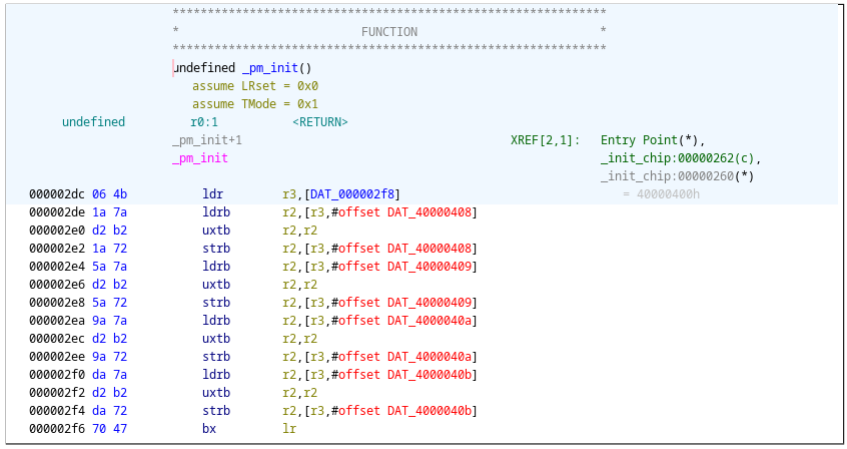

Fig. 7. *Microchip MPLabX* bootstrap code for SAMD21 Power Manager controller

The previously implemented *GCLK* core peripheral could also have been supported by this default memory region, but we may need to react to some read or write operations in the future if we want or need to support multiple clock sources (but for now our simple solution is enough to do the job).

We now have implemented our core peripherals. Fig **??** shows our progression in our journey to support this SAMD21 MCU.

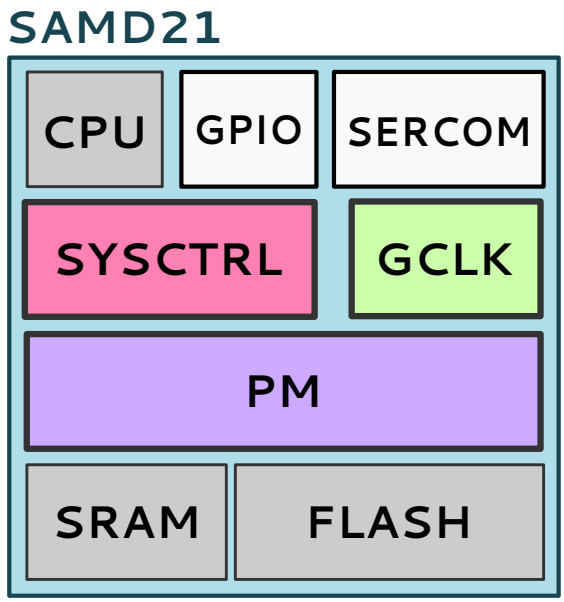

Fig. 8. SAMD21 MCU overview, core peripherals done

8 Step 5: implementing SERCOM peripherals

SAMD21's *SERCOM* (stands for *SER*ial *COM*munication) peripheral provides an interface for different serial 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.

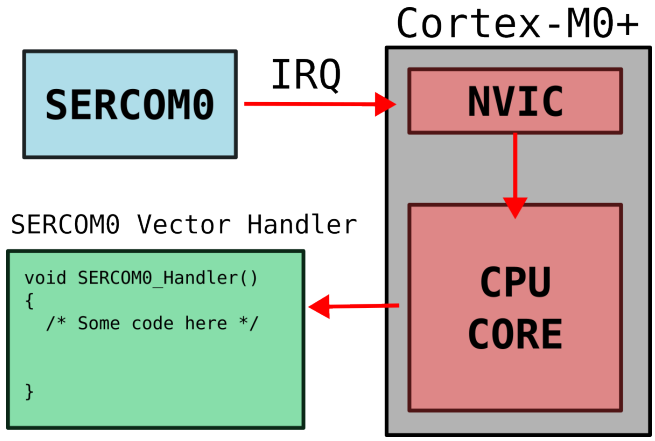

Fig. 9. From *SERCOM0* IRQ to handler execution

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 9). 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 10 taken from the datasheet shows the interrupted lines associated with each *SERCOM* peripheral.

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

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

Fig. 10. *SERCOM* peripherals NVIC interrupt lines as specified in the datasheet

USART interface to be available to the user in a terminal. Instead of outputting bytes to some GPIOs, we will forward 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.

SERCOM implementation is located in `hw/misc/samd21_sercom.c` as this peripheral supports both USART and SPI protocols. Usually, any USART (or UART) peripheral implementation is placed in `hw/char/` while SPI-related peripherals are located in `hw/ssi/`. It implements a new object type `TYPE_SAMD21_SERCOM` as described in previous sections of this document. We will not explain in this section how this new type is declared and initialized, the reader is invited to read the corresponding source code with a focus on the *samd21_sercom_class_init()* and *samd21_sercom_init()* functions.

8.1 Adding support for USART protocol

When a byte is sent through the *SERCOM* peripheral, we simply tell QEMU that the associated *character device* has received one byte of data (listing 37). 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) by setting the peripheral interrupt flags (listing 38). The corresponding interrupt request (*IRQ*) is triggered by our code if at least one event has been enabled by software (listing 39), by calling *qemu_set_irq()*.

When `qemu_set_irq()` is called with a level of 1, the corresponding NVIC interrupt line is triggered and this causes the associated vector handler to

```
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;
}
```

Listing 37. SERCOM USART sending function

```
1  static void uart_receive(void *opaque, const uint8_t *buf, int
   ↪  size)
2  {
3    SAMD21SERCOMState *s = SAMD21_SERCOM(opaque);
4    int i;
5
6    /* Sanity checks (exit if no byte received or RX FIFO is
     ↪  full). */
7    if (size == 0 || s->rx_fifo_len >= UART_FIFO_LENGTH) {
8      return;
9    }
10
11   /* Load received bytes into RX FIFO. */
12   for (i = 0; i < size; i++) {
13     uint32_t pos = (s->rx_fifo_pos + s->rx_fifo_len) %
       ↪  UART_FIFO_LENGTH;
14     s->rx_fifo[pos] = buf[i];
15     s->rx_fifo_len++;
16   }
17
18   /* Set RXC bit (receive complete) in INTFLAG register. */
19   s->reg8[R_SERCOM_INTFLAG] |= (1 <<
     ↪  R_SERCOM_INTFLAG_RXC_SHIFT);
20
21   /* Trigger IRQ if required. */
22   samd21_update_irq(s);
23 }
```

Listing 38. SERCOM UART incoming data processing

```
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);
}
```

Listing 39. SAMD21 SERCOM IRQ update

```
static Property samd21_sercom_properties[] = {
        DEFINE_PROP_CHR("chardev", SAMD21SERCOMState, chr),
        DEFINE_PROP_END_OF_LIST(),
};
```

Listing 40. SAMD21 SERCOM peripheral properties

```
static void samd21_sercom_realize(DeviceState *dev, Error
↪  **errp)
{
        SAMD21SERCOMState *s = SAMD21_SERCOM(dev);

        /* Set QEMU character device backend callbacks to handle
        ↪  UART events. */
        qemu_chr_fe_set_handlers(&s->chr, uart_can_receive,
        ↪  uart_receive,
        uart_event, NULL, s, NULL, true);
}
```

Listing 41. SAMD21 SERCOM chardev configuration

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.

We also use a specific object property named `chardev` mapped to our `SAMD21SercomState` structure member `chr` to allow external code to assign a character device to use for input/output (listing 40). This same `chr` structure member is used in our code to read incoming data typed in by the user and write data sent by the firmware back into the user's terminal. This character device is configured in *samd21_sercom_realize()* (listing 41).

8.2 Adding support for SPI master

The SAMD21 SERCOM peripheral can also acts as an *SPI* master or slave, depending on its configuration. Luckily for us, QEMU provides a way to create a system bus dedicated to synchronous serial interfaces, or *SSI*. Creating a dedicated *SSI* bus is required as we may want to connect multiple devices on it, each device being selected through a dedicated *Chip Select* line (usually called

```
struct SAMD21SERCOMState {
  SysBusDevice parent_obj;

  MemoryRegion iomem;
  CharBackend chr;
  SSIBus *ssi;

  /* ... */
```

Listing 42. SERCOM state structure with SSI reference

```
/* Initialize a new SSI bus. */
s->ssi = ssi_create_bus(dev, "ssi");
```

Listing 43. SERCOM SSI bus creation

CS). We will cover how *CS* lines are handled later in this document. First, we need to update our SERCOM state structure to include a reference to an *SSI* bus that is going to be dynamically allocated (listing 42). Then, we add in our *sercom_init()* function a few lines to create our bus (*SSIBus*) and save it inside our state structure (listing 43).

Once this bus created, we can handle data transmission when a write operation is performed on the *DATA* MMIO register. First, we create a dedicated function (listing 44) in charge of transferring the content of the *DATA* register to our *SSI* bus, and we call this function every time a byte is written into the *DATA* regiser (listing 45). Since *SPI* is full-duplex, we send and receive data at the same time, so we end up updating the *DATA* register each time a byte has been sent.

Our SERCOM peripheral has now its own *SSI* bus and is able to send data to and receive data from it. We will later connect an emulated *SPI* Flash chip to one of our SERCOM peripheral in order to communicate with it.

Last but not least, we add some initialization code in the SAMD21 instance initialization callback *samd21_realize()* in order to create our 6 SERCOM interfaces, and assign the first one to our default serial interface (listing 46).

8.3 Creating our default 6 SERCOM interfaces

Once our *SERCOM* object type implemented, we can add the 6 default *SERCOM* interfaces to our SAMD21 *MCU*. We first add an array of

```
static void sercom_spi_transfer(SAMD21SERCOMState *s)
{
  /* Send and receive byte. */
  s->reg16[R_SERCOM_DATA] = ssi_transfer(s->ssi,
  ↪  s->reg16[R_SERCOM_DATA] & 0xFF);

  /* Set DRE, RXC and TXC flag in the INTFLAG register, to tell
  ↪  the software that
  more bytes can be sent after this function call. */
  s->reg8[R_SPI_INTFLAG] |= (1 << R_SPI_INTFLAG_DRE_SHIFT);
  s->reg8[R_SPI_INTFLAG] |= (1 << R_SPI_INTFLAG_TXC_SHIFT);
  s->reg8[R_SPI_INTFLAG] |= (1 << R_SPI_INTFLAG_RXC_SHIFT);
}
```

Listing 44. SERCOM SPI byte transfer

```
  case A_SERCOM_DATA:
  {
    switch (s->current_mode)
    {
      /* ... */

      case SERCOM_SPI:
      {
        /* Write register. */
        s->reg16[R_SERCOM_DATA] = (value & 0xffff);

        /* Transfer through SSI. */
        sercom_spi_transfer(s);
      }
      break;

      default:
        break;
      }
    }
    break;
```

Listing 45. Handling SPI byte send

```
for (i=0; i<6; i++)
{
  if (!sysbus_realize(SYS_BUS_DEVICE(&s->sercom[i]), errp)) {
    return;
  }

  /* We retrieve our SERCOM peripheral MMIO region
     into `mr`. */
  mr = sysbus_mmio_get_region(
    SYS_BUS_DEVICE(&s->sercom[i]),
    0);

  /* And we map it to the correct base address inside
     our container.  */
  memory_region_add_subregion_overlap(&s->container,
    SAMD21_SERCOM0_BASE + 0x400*i, mr, 0);

  /*
  * Last but not least, we need to connect our SERCOM IRQ
  * line to the corresponding NVIC input IRQ line, using
  * the `sysbus_connect_irq` function. We need to know the
  * IRQ line number used in the SERCOM peripheral
  * implementation (again, identified by a sequential
  * number that is incremented in the exact order these
  * IRQ lines have been created) and connect it to the
  * correct GPIO exposed by the CPU NVIC.
  * Here we are connecting SERCOM's IRQ line #n to NVIC
  * input GPIO #9+n, as described in the SAMD21
  * documentation (section 11.2.2).
  */
  sysbus_connect_irq(SYS_BUS_DEVICE(&s->sercom[i]), 0,
    qdev_get_gpio_in(DEVICE(&s->cpu), 9 + i));
}
```

Listing 46. SAMD21 SERCOM interfaces initialization

```
for (i=0; i<6; i++)
{
  /* Create an instance of SERCOM peripheral. */
  object_initialize_child(obj, "sercom[*]", &s->sercom[i],
    TYPE_SAMD21_SERCOM);

  /* Special case for SERCOM0: we want to add an alias
     'serial0' for it. */
  if (i == 0)
  {
    object_property_add_alias(obj, "serial0",
      OBJECT(&s->sercom[i]), "chardev");
  }
}
```

Listing 47. SAMD21 SERCOM interfaces creation

`SAMD21SercomState` structures in our `SAMD21State` structure, initialize 6 interfaces in *samd21_init()* (listing 47) and then configure all of them in *samd21_realize()* (listing 48). For each interface, we map the corresponding memory region into our container at the expected address using *memory_region_add_subregion_overlap()* and then connect each *SERCOM* IRQ line to the CPU NVIC input GPIO following the datasheet. Interrupt numbers 9 to 15 are connected to our 6 different *SERCOM* interfaces. We create a new property `serial0` for our SAMD21 *MCU* that is an alias of *SERCOM0* `chardev` property.

SAMD21 *SERCOM* peripherals are now supported, Fig **??** shows our progress in our journey to add support for SAMD21 into QEMU.

```
for (i=0; i<6; i++)
{
  if (!sysbus_realize(SYS_BUS_DEVICE(&s->sercom[i]), errp)) {
    return;
  }

  /* We retrieve our SERCOM peripheral MMIO region into `mr`. */
  mr = sysbus_mmio_get_region(SYS_BUS_DEVICE(&s->sercom[i]), 0);

  /* And we map it to the correct base address inside our
     container. */
  memory_region_add_subregion_overlap(&s->container,
                  SAMD21_SERCOM0_BASE + 0x400*i, mr, 0);

  /* We connect the SERCOM IRQ #0 to our corresponding
     NVIC GPIO. */
  sysbus_connect_irq(SYS_BUS_DEVICE(&s->sercom[i]), 0,
    qdev_get_gpio_in(DEVICE(&s->cpu),
    9 + i));
}
```

Listing 48. SAMD21 SERCOM interfaces configuration

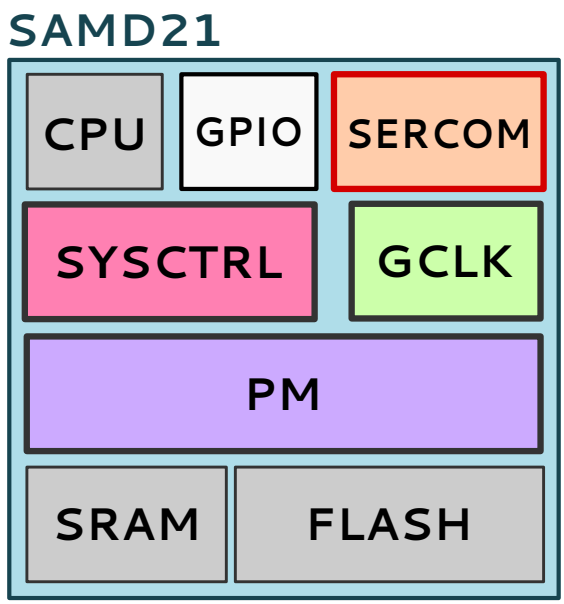

Fig. 11. SAMD21 MCU overview, core peripherals and SERCOM done

9 Step 6: implementing SAMD21 GPIO peripheral

The SAMD21 *MCU* provides a dedicated peripheral to interact with the *MCU* physical input /ouput lines or *GPIO*s (for *G*eneral *P*urpose *I*nput/*O*utput). Emulating *GPIO*s in QEMU may seem pointless but since these *GPIO*s may be connected to other emulated devices, it makes sense to get it supported in our *MCU* implementation. Especially because our custom board also hosts an SPI Flash chip that is connected to a SAMD21 *MCU* through an SPI interface and a dedicated GPIO line (used for SPI chip selection).

Implementation is really no different from what we have done so far and follows the same logic:

1. We create a dedicated source file (and header file) in which we define a new object type
2. We populate our new object type state structure and properties with the required members
3. We handle read/write operations performed on the peripheral MMIO registers

9.1 Creating a new object type and associated source file

We create a new file named `samd21_gpio.c` in `hw/gpio/` and its associated header file in `include/hw/gpio/samd21_gpio.h`, and implement the basic code to create a new type named `TYPE_SAMD21_GPIO`.

One of the main interesting characteristic of the state structure associated to this peripheral is the use of `qemu_irq`, which is a type dedicated to IRQs but that will be used in our case as outputs that can be connected to any other object input GPIO. This mechanism will allow to propagate a GPIO state to other components of the board and make them react to these changes.

The state structure (listing 49) contains multiple members to handle the peripheral operations and associated registers. As for our previous hardware peripherals, we define a memory region (referenced by `mmio` in our structure) to handle any read and write operation on the peripheral MMIO registers and update the object state accordingly.

9.2 Handling read and write operations on MMIO registers

We set up a dedicated memory region with its read/write operation handlers referenced in a `MemoryOps` structure (listing 50) and implemented two handlers: *samd21_port_read()* for read operations and *samd21_port_write()* for write operations.

```
struct SAMD21GPIOState {
  SysBusDevice parent_obj;

  MemoryRegion mmio;
  qemu_irq irq;


  uint32_t out;
  uint32_t old_out;
  uint32_t old_out_connected;

  uint32_t in;
  uint32_t in_mask;
  uint32_t dir;
  uint32_t control;
  uint32_t cnf[SAMD21_PORT_PINS];

  qemu_irq output[SAMD21_PORT_PINS];
  qemu_irq detect;
};
```

Listing 49. SAMD21 GPIO peripheral state structure

```
static const MemoryRegionOps gpio_ops = {
        .read =  samd21_port_read,
        .write = samd21_port_write,
        .endianness = DEVICE_LITTLE_ENDIAN,
        .impl.min_access_size = 4,
        .impl.max_access_size = 4,
};
```

Listing 50. SAMD21 GPIO peripheral MMIO operations

```
1  static void update_output_irq(SAMD21GPIOState *s, size_t i,
2  bool connected, bool level)
3  {
4    /* If pin is not connected, reflect its floating state by
5       setting the IRQ level to -1. */
6    int64_t irq_level = connected ? level : -1;
7    bool old_connected = extract32(s->old_out_connected, i, 1);
8    bool old_level = extract32(s->old_out, i, 1);
9
10   /* If output has been disconnected/connected or level
11      changed, notify. */
12   if ((old_connected != connected) || (old_level != level)) {
13     qemu_set_irq(s->output[i], irq_level);
14   }
15
16   /* Save current pin state. */
17   s->old_out = deposit32(s->old_out, i, 1, level);
18   s->old_out_connected = deposit32(s->old_out_connected, i,
19                                    1, connected);
20 }
```

Listing 51. SAMD21 GPIO update

This GPIO peripheral provides an optional pull-up resistor that can be programmatically set, we need to take this into account when the register (*IN*) holding our input values is read. Writing to registers *OUTSET*, *OUTCLR* or *OUTTGL* cause the *OUT* register to be modified accordingly. Setting a bit in *OUTSET* causes the same bit to be forced in the *OUT* register while setting a bit in *OUTCLR* clears the corresponding bit in *OUT*. Setting a bit in *OUTTGL* toggles the corresponding bit in *OUT*. This behavior is implemented in *samd21_port_write()*.

Whenever the *OUT* register value changes we need to update our object *GPIO*s, which is implemented in *update_output_irq()* (listing 51). We track each output individually and update the corresponding IRQ when a change is observed.

9.3 Adding our GPIO peripheral to SAMD21

Once this *GPIO* peripheral implemented, we can use it in our SAMD21 *MCU* implementation. As usual, we start by adding a new member `port` of

type `SAMD21GPIOState` into our `SAMD21State` structure and initializing it in *samd21_init*:

```
/*
 * Initialize our PORT (GPIO) controller.
 */
object_initialize_child(obj, "port", &s->port,
  TYPE_SAMD21_PORT);
```

And we realize it in *samd21_realize()* and map its MMIO memory region at the correct address (listing 52).

```
/*
* Initialize PORT controller.
*/
if (!sysbus_realize(SYS_BUS_DEVICE(&s->port), errp)) {
        return;
}

/* We retrieve our PORT controller MMIO region into `mr`. */
mr = sysbus_mmio_get_region(SYS_BUS_DEVICE(&s->port), 0);

/* And we map it to the correct base address inside our
↪  container. */
memory_region_add_subregion_overlap(&s->container,
↪  SAMD21_PORT_BASE, mr, 0);
```

Listing 52. SAMD21 GPIO initialization

SAMD21 now supports all our required peripherals as shown in Fig 12, we have everything we need to start emulating a compatible firmware. Note that some other hardware peripherals are still missing, but since they are not critical for our target firmware they have been left unimplemented. These peripherals can be added by following the same methodology, based on the SAMD21 datasheet.

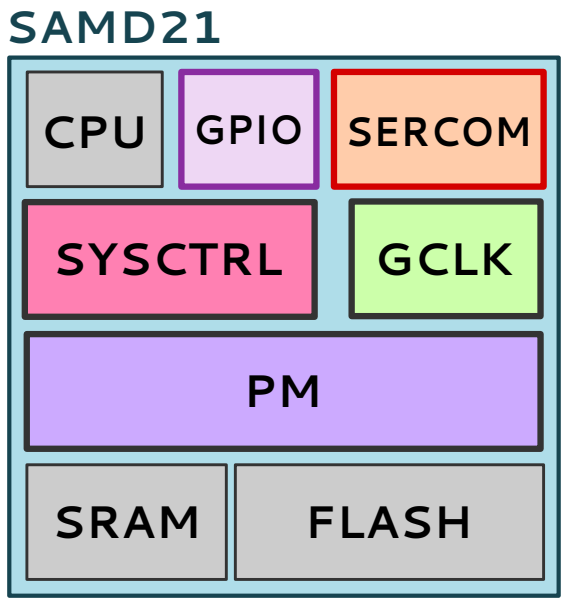

Fig. 12. SAMD21 MCU overview, all our target peripherals are now supported

10 Step 7: putting everything together

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.

10.1 Connecting the QEMU console to our *SERCOM0* interface

First, we need to assign our machine first serial interface to our SAMD21 *MCU*'s `serial0` character device (listing 53). QEMU *serial_hd()* function retrieves the first serial device configured for our custom board and assign it to our *SERCOM0* interface, allowing the user to interact with it through a terminal.

```
/**
 * We then set the MCU "serial0" property in order to use the
 * first defined serial hardware device specified in command
 * line.
 */
qdev_prop_set_chr(DEVICE(&s->samd21), "serial0", serial_hd(0));
```

Listing 53. Qb Lil'board serial interface initialization

10.2 Creating and plugging an external SPI Flash

Once our MCU and its interfaces initialized, we can create an SPI Flash device and connect it to our MCU's SERCOM4 interface similarly to our real custom board (listing 54). This flash device will be linked to a character device which content will be used as this Flash chip content, and this behavior is implemented off-the-shelf in QEMU's M25P80 Flash chip emulation code. This flash chip will be created and connected to SERCOM4 if and only if at least one device drive is provided through QEMU command-line arguments.

We use QEMU's *qdev_new()* to dynamically create a specific device, in our case an *SST25VF016B* Flash chip, and we connect it to SERCOM4's interface SSI bus with a call to *qdev_realize_and_unref()*. Last but not least, we also need to tie this chip *CS* line to a dedicated *GPIO* (#21) of our SAMD21 MCU with a call to *qdev_connect_gpio_out()*.

```
/* Look for a block device that represents our SPI flash device
↪  content. */
di = drive_get(IF_NONE, 0, 0);
blk = di ? blk_by_legacy_dinfo(di) : NULL;
if (blk != NULL)
{
  /* Create an SPI flash device and attach to
     SERCOM4 SSI interface. */
  dev = qdev_new("sst25vf016b");
  qdev_prop_set_drive(dev, "drive", blk);
  qdev_prop_set_uint8(dev, "cs", 0);
  qdev_realize_and_unref(
  dev, BUS(s->samd21.sercom[4].ssi), &error_fatal);

  /* Connect Flash CS line to our GPIO. */
  cs_line = qdev_get_gpio_in_named(dev, SSI_GPIO_CS, 0);
  qdev_connect_gpio_out(DEVICE(&s->samd21.port), 21, cs_line);
}
```

Listing 54. Flash creation

```
./qemu-system-arm -M qb-lilboard --kernel /tmp/QemuTest.ino.elf
↪  -drive file=flash.bin,if=none,format=raw -serial stdio
  ___  _        _      _ _ _ _____                          _
 / _ \| |__    | |    (_) ( ) __ )  ___    __ _ _ __ __| | |
| | | | '_ \   | |    | | |/|  _ \ / _ \ / _` | '__/ _` |
| |_| | |_) |  | |___| | | | |_) | (_) | (_| | | | (_| |
 \__\_\_.__/   |_____|_|_| |____/ \___/ \__,_|_|  \__,_|

Loading 16 bytes from SPI Flash ...
HEX>>> 48 65 6C 6C 6F 20 57 6F 72 6C 64 20 21 0 0 0
CHR>>> H  e  l  l  o     W  o  r  l  d     !

Enter some text> test
> You wrote: test
Enter some text>
```

Listing 55. Testing firmware

11 Debugging our custom board

This section will cover how the previously defined emulated board can be used for debugging and vulnerability research. We use a test firmware to check that everything is working fine.

11.1 Testing UART and SPI communication

First, we write a small program designed to run on this board that will read the first 16 bytes of the external Flash memory, compile it and run it with QEMU (listing 55).

We provide QEMU with a *flash.bin* binary file containing the content of Flash memory (starting with `Hello World !`) and emulate the firmware, which reads the first 16 bytes from the emulated Flash memory and return them on its main UART interface (*SERCOM0*). Everything works as expected as the SPI interface operates properly and the user can interact with the firmware's UART interface through the terminal.

11.2 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
```

Listing 56. Emulating our firmware with GDB server enabled

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
[...]
(gdb) c
Continuing.
```

Listing 57. Debugging with *gdb-multiarch*

12 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.

13 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 provides 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 QEMU Project. *The QEMU Object Model (QOM)*. URL: https://qemu-project.gitlab.io/qemu/devel/qom.html.

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

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

[11] 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.

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