The DNA of Event Driven Embedded Applications

Hugo Pristauz, Ignacio Lopez Negrete – Feb. 2023 / Bluenetics GmbH

Abstract

Clean software architecture concepts [1,2] have been introduced to enhance the efficiency of software development and to keep software readable and understandable in order to allow efficient and error free modifications and extensions over the release cycles of software.

To support *clean software-development* we propose strictly modular software architectures for *event driven embedded applications* and introduce the concept of *interactions* to represent any kind of interactions between modules. On top of such *interactions,* we first formulate the concept of *interaction sequences.* Based on those we introduce the *interaction profile*, the key concept of this paper. As a combination of the *module list* and set of all *interaction sequences* of an application, it is a *high-level description* of the application.

The *interaction profile* acts like a DNA of an event driven embedded application. It allows to extract *interaction interfaces*, *interaction sequence charts*, *module interfaces, interaction graphs,* the *system graph* and the *system interaction chart*.

After presenting different kinds of translating an *interaction profile* into C-code we highlight the benefits of a *pluggable module architecture*. Last, but not least, we demonstrate how C-coded public module interfaces, as well as skeleton C-code for module implementations can be automatically generated from the *interaction profile*.

A Toy Application

Let us begin our study with a *toy application,* which is running in an embedded environment. The term *toy application* refers to some plain sample application which still allows to pinpoint common problems of embedded software architectures. Let us assume an embedded playground represented by a *Nordic nRF52 DK (development kit)* [3]. Such board supports 4 buttons (*button @1*, ..., *button @4*), 4 LEDs (LED @1, ..., LED @4) and wireless connectivity like BLE or Bluetooth mesh (figure 1).

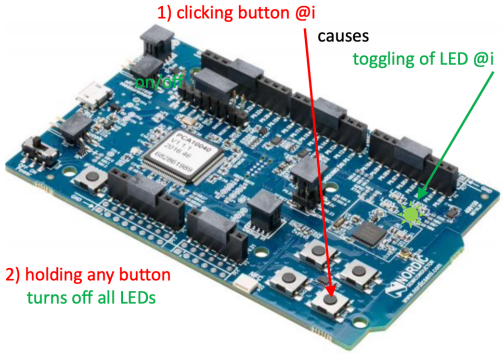

Figure 1: Playground based on nRF52 DK

A definition for our *toy application* can be phrased as follows:

1) clicking button @i causes toggling of LED @i (where i ranges from 1 to 4)
2) holding any button turns off all LEDs

Efficient Software Development Process

For event driven embedded applications we propose the 3-phase SW development process of figure 2:

1) Split up any embedded event driven application into modules and organize them as a layered module architecture [4].

2) Define module interfaces and provide a high- level description of module interactions. The outcome of this phase should be sharable in understandable form with all team members.

3) Implementation of specific modules. Each module should be treated as an independent sandbox, which communicates with the outside world only via the public module interface.

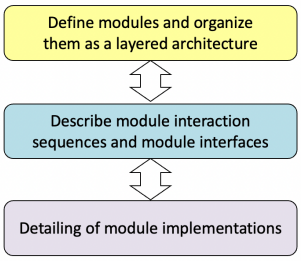

Figure 2: Efficient software development process

We trust the reader to agree with the benefits of a modular software architecture. The use of a layered module architecture is very common where, e.g., top-level application

logic and user interface control are in high-level layers, while wireless communication tasks or driver functionality are found in low-level layers.

Our opinion is that the last step of this development process, the module implementation, is usually easier and less challenging than phase 2, related to a smaller complexity of the surrounding environment (nevertheless we may notice that, e.g., the implementation of some driver module or mesh model might also have its own challenges).

In our experience the most crucial task deals with a high-level *description of the module interactions* and the required *interface definitions*, as this step covers the overall system behavior and thus affects the whole software team. In addition, we are facing the challenge of delivering an easy understandable outcome of this phase, which can be shared with all team members. Once the overall system can be understood in its full extent on an abstract level, the missing details can be investigated and developed step by step.

In this paper we propose an approach to master this challenge by defining a set of module *interaction sequences*, which can be visualized by *interaction sequence charts*, a graphical representation closely related to *message sequence charts* [5].

Modular Architecture

Let us demonstrate this process by applying phase 1, splitting up our toy application into modules in order to implement the *layered architecture* [4] shown in figure 3:

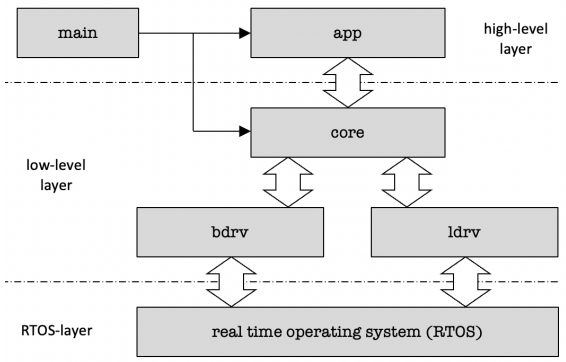

Figure 3: Modular layered architecture of toy app

Such architecture comprises:

- A `main` program setting up a processing environment by initializing *high-level* and *low-level* layers.
- An `app` module, which implements the application logic by processing button events from a `core` module, which are translated to `core` commands turning a particular LED @i either on or off
- A `core` module which mediates between two driver modules and the `app` module
- A button driver module (`bdrv`) which interacts with the `core` module triggered by button events
- An LED driver module (`ldrv`) translating LED control commands from the `core` module into on/off states of the LEDs
- The real time operating system (`RTOS`), which forms the bottom layer.

The reader might argue that a split-up of our tiny *toy application* into so many different modules is an overkill, but our main intention here is to demonstrate efficient dealing with *interactions* within modular applications.

Module Interactions

Step 2 of the proposed software development process in figure 2 deals with the definition of *module interfaces* and the description of (*module-*) *interactions*. In general, *interactions* between modules can have different forms, such as

- calling a function of another module
- invoking a registered (or hard coded) callback of another module
- changing the value of a data structure of another module
- reading the value of a data structure of another module
- sending an event message to another module

In an object oriented context we might further talk about

- calling a method of an object instance which is declared in another module
- changing or reading a data member of an object instance which is part of another module

We introduce the notion of a uniform *interaction*, which shall represent any kind of above listed module interactions.

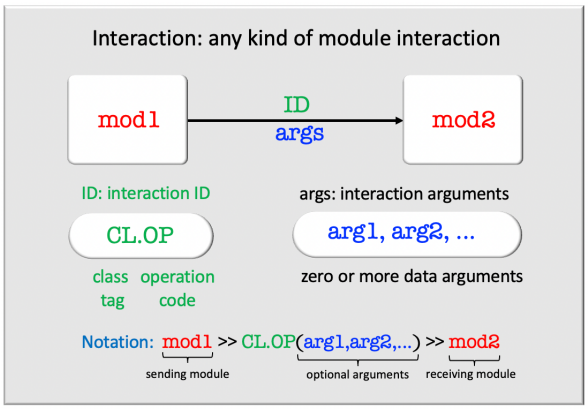

Figure 4: Concept of an *interaction*

The concept of an *interaction* is very simple: we use an *interaction ID* which characterizes the kind of *interaction*, followed by a (possibly empty) list of *interaction arguments*. We call such combination the *interaction prototype*. Consider now a conventional function call

```
led_set(i,0);  // turn LED @i off
```

The function name `led_set` may be considered as the *interaction ID*, and `(i,0)` as the *interaction argument* list. The subtle thing here is that the *interaction ID* of our introduced *interaction* concept *has* to comprise two parts:

- an *interaction class tag*
- an *interaction operation* code

Thus a method call of the following C++ statement, where `led` is a class instance and `set` is the name of a class method, would come closer to our intentions:

```
led.set(i,0);  // turn LED @i off
```

The benefit of such composite interaction ID concept is that it allows to group the total plurality of *interactions* into a set of *interaction classes*, and to define *interaction interfaces* for each *interaction class*. Let us demonstrate this approach with some examples.

Example: LED Interaction Class

As an example consider an *interaction class* to describe all module interactions for controlling the on/off state of LEDs supported by the following interactions:

```
LED.SET(i,on)    // turn LED @i on/off
LED.OFF()        // turn all LEDs off
```

If a module is capable of supporting (either sending and/or receiving) some *interaction class* then we say that the module provides an *interaction interface* for the supported *interaction class*. Such kind of *interaction interface* can be represented in graphical form (figure 5).

```
      +--------------------+
      |        LED:        | LED interface
SET --|       @i,on        | turn LED @i on/off
OFF --|                    | turn all LEDs off
      +--------------------+
```

Figure 5: `LED` *interaction interface*

Note that the notation `@i` is used here to emphasize that `i` represents an instance index, which is used, e.g., for addressing button and LED instances.

Example: BUTTON Interaction Class

In a similar way we can introduce a `BUTTON` *interaction class* and a related BUTTON *interaction interface* supporting the *interactions*:

```
BUTTON.CLICK(i,n)  // button @i n x clicked
BUTTON.HOLD(i,ms)  // button @i held over ms
```

This *interaction class* is related to the graphical representation of the BUTTON *interaction interface* shown in figure 6.

```
       +--------------------+
       |       BUTTON:      | BUTTON interface
CLICK --|       @i,n        | button @i n x clicked
 HOLD --|       @i,ms       | button @i held over ms
       +--------------------+
```

Figure 6: BUTTON interaction interface

Example: SYS Interaction Class

Finally let us introduce a SYS (system) interaction class. Remember that any kind of module interaction should be represented by an *interaction*. Consequently this applies also to module initialization.

```
SYS.INIT()  // init module
```

Figure 7 shows the related graphical representation of the SYS *interaction interface*.

```
       +--------------------+
       |        SYS:        | SYS interface
INIT --|                    | init module
       +--------------------+
```

Figure 7: SYS *interaction interface*

Interaction Sequence

Consider now the initialization process of the application, which we want to run as follows:

- Module `main` should first initialize the low-level layer by initializing module core, which subsequently shall initialize the button driver (`bdrv`) and the LED driver (`ldrv`).
- After the *low-level* layer is initialized, module `main` should initialize the *high-level* layer by initializing module `app`.

Such kind of specification can be translated into a so-called *interaction sequence* as follows:

```
Sequence(Init_Sequence)
{
  main >> SYS.INIT() >> core;        // init module
  core >> SYS.INIT() >> bdrv;
  core >> SYS.INIT() >> ldrv;
  main >> SYS.INIT() >> app;
}
```

Figure 8*: interaction sequence* `Init_Sequence`

Interaction Profile

We are now at the point to introduce the important concept of an *interaction profile*, which is the key concept of this paper. Let us combine the *list of all modules* with the *set of all interaction sequences* and call this construct the *interaction profile* of the application (figure 9).

```
Module(main); // main entry point: init core, app
Module(app);  // high level logic
Module(core); // HW core module, managing drivers
Module(bdrv); // button driver
Module(ldrv); // LED driver

Sequence(Init_Sequence)
{
  main >> SYS.INIT() >> core;         // init module
  core >> SYS.INIT() >> bdrv;
  core >> SYS.INIT() >> ldrv;
  main >> SYS.INIT() >> app;
}

Sequence(Toggle_Sequence)
{
  Action(bdrv,"click button @i");
  bdrv >> BUTTON.CLICK(i,n) >> core;  // button @i n x clicked
  core >> BUTTON.CLICK(i,n) >> app;
  Action(app,"on = state[i] = !state[i]");
  app >> LED.SET(i,on) >> core;       // turn LED @i on/off
  core >> LED.SET(i,on) >> ldrv;
  Action(ldrv,"toggle LED @i");
}

Sequence(Off_Sequence)
{
  Action(bdrv,"hold button @i");
  bdrv >> BUTTON.HOLD(i,ms) >> core; // button @i held over ms
  core >> BUTTON.HOLD(i,ms) >> app;
  Action(app,"turn all LEDs off; state[1..4] = 0");
  app >> LED.OFF() >> core;          // turn all LEDs off
  core >> LED.OFF() >> ldrv;
  Action(ldrv,"turn all LEDs off");
}
```

Figure 9*: interaction profile* of toy app

Assume now that the *interaction profile* is the outcome of phase 2 in the proposed software development process, trusting to have developed a comprehensive high-level description of our toy application. How comprehensive is it? Which kind of information can be extracted from the *interaction profile*?

Let us identify all *interactions* and make a list of *interaction prototypes* (without listing the same *interaction prototype* multiple times) and group them by *interaction class*.

```
SYS.INIT()              // init module

LED.SET(i,on)           // turn LED @i on/off
LED.OFF()               // turn all LEDs off

BUTTON.CLICK(i,n)       // button @i n x clicked
BUTTON.HOLD(i,ms)       // button @i held over ms
```

We identify three interaction classes: `SYS`, `LED` and `BUTTON`. Can we reconstruct the related *interaction interfaces* from the *interaction profile*? Sure – there is not any difficulty to reconstruct figures 5,6 and 7 from the list of *interaction prototypes* presented before.

Can we extract the list of modules and the list of interaction sequences? Sure – this is how the interaction profile is constructed. Let us summarize:

An interaction profile allows to extract

- the list of all *modules*
- all *interaction sequences*
- all *interaction interfaces*

Interaction Sequence Charts

Interaction sequence charts are a graphical representation of *interaction sequences*. They are closely related to *message sequence charts*, which are frequently used as an analysis or documentation tool for communication systems [5]. While *message sequence charts* are focused on message transmissions we generalize this concept (more or less unmodified) by replacing *messages* with *interactions*.

An *interaction sequence chart* of the *interaction sequence* `Init_Sequence` (figure 8) is shown in figure 10.

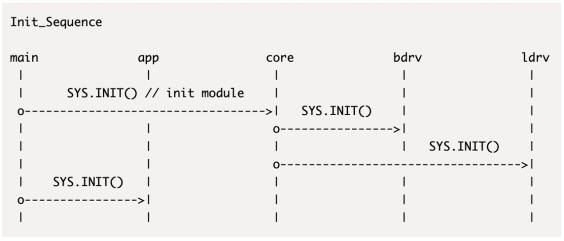

Figure 10*: interaction sequence chart of* `Init_Sequence`

Due to its graphical nature *interaction sequence charts* give a very intuitive high-level overview of what is going on inside of an application, and in which time order. Let us present also the remaining two *interaction sequences* in terms of *interaction sequence charts* (figure 11,12).

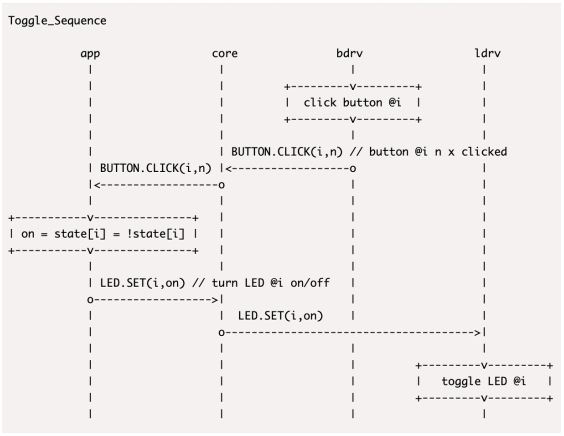

Figure 11*: interaction sequence chart of* `Toggle_Sequence`

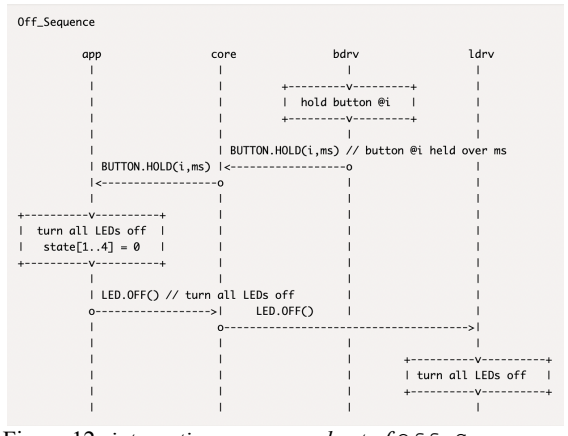

Figure 12*: interaction sequence chart of* `Off_Sequence`

Even we would know nothing about the application, the study of these charts allows us to conclude as follows:

- The application consists of five modules (`main`, `app`, `core`, `bdrv` and `ldrv)` and there are three *interaction sequences* provided.
- The first sequence (`Init_Sequence`) defines the initializing process: initiator is the `main` module, which initializes first the `core` module and after that the `app` module. The `core` module acts obviously as the 'manager' of the button driver and LED driver modules (`bdrv`, `ldrv`), since it takes care of initializing these modules, once triggered by `main`.
- The second sequence (`Toggle_Sequence`) starts at the button driver (`bdrv`), initiating an *interaction* `BUTTON.CLICK(i,n)` addressed to `core`, which forwards it to `app`. The `app` module translates the *interaction* into `LED.SET(i,on)` addressed to `core`, which delegates it to the `LED` driver, where it is processed in order to turn LED @i either on or off.
- A similar sequence (`Off_Sequence`) is defined for a scenario to turn off all LEDs if any button is pressed and held. We leave the investigation of the details to the reader.

Module Interfaces

But we can extract even more information from the *interaction profile*. Considering all the different *interaction sequences* we can ask the question, which *interaction interfaces* must a specific module support in order to be capable of participating in all involved module interactions.

Let us demonstrate this idea for module `app`, which is involved in all three *interaction sequences*.

- `Init_Sequence`: module `app` receives a `SYS.INIT` interaction from module `main`, thus the module has to support the `SYS` *interaction interface*.
- `Toggle_Sequence`: module `app` receives a `BUTTON.CLICK` interaction from `core` and addresses an `LED.SET` interaction to `core`, thus the *interaction interfaces* `BUTTON` and `LED` must be supported.
- `Off_Sequence`: module `app` receives a `BUTTON.HOLD` message from `core` and addresses an `LED.OFF` interaction to `core`, thus the *interaction interfaces* `BUTTON` and `LED` must be supported.

Bringing all these requirements together we can construct the following graphical description of a *module interface* for `app` (figure 13).

```
                +--------------------+
                |        app         | high level logic
                +--------------------+
                |        SYS:        | SYS interface
 main -> INIT ->|                    | init module
                +--------------------+
                |       BUTTON:      | BUTTON interface
core -> CLICK ->|       @i,n         | button @i n x clicked
 core -> HOLD ->|       @i,ms        | button @i held over ms
                +--------------------+
                |        LED:        | LED interface
  core <- SET <-|      @i,onoff      | turn LED @i on/off
  core <- OFF <-|                    | turn all LEDs off
                +--------------------+
```

Figure 13*: module interface* of module `app`

In a similar way we can extract the module interface of the remaining modules.

```
                +--------------------+
                |        main        | main entry point: init core, app
                +--------------------+
                |        SYS:        | SYS interface
app,core <- INIT <-|                | init module
                +--------------------+
```

Figure 14*: module interface* of module `main`

```
                 +--------------------+
                 |        core        | HW core module, managing drivers
                 +--------------------+
                 |        SYS:        | SYS interface
     main -> INIT ->|                | init module
bdrv,ldrv <- INIT <-|                | init module
                 +--------------------+
                 |       BUTTON:      | BUTTON interface
    bdrv -> CLICK ->|       @i,n       | button @i n x clicked
     bdrv -> HOLD ->|       @i,ms      | button @i held over ms
     app <- CLICK <-|       @i,n       | button @i n x clicked
      app <- HOLD <-|       @i,ms      | button @i held over ms
                 +--------------------+
                 |        LED:        | LED interface
       app -> SET ->|      @i,onoff     | turn LED @i on/off
       app -> OFF ->|                   | turn all LEDs off
      ldrv <- SET <-|      @i,onoff     | turn LED @i on/off
      ldrv <- OFF <-|                   | turn all LEDs off
                 +--------------------+
```

Figure 15*: module interface* of module `core`

```
                +--------------------+
                |        ldrv        | LED driver
                +--------------------+
                |        SYS:        | SYS interface
 core -> INIT ->|                    | init module
                +--------------------+
                |        LED:        | LED interface
  core -> SET ->|      @i,onoff      | turn LED @i on/off
  core -> OFF ->|                    | turn all LEDs off
                +--------------------+
```

Figure 16*: module interface* of driver module `ldrv`

```
                +--------------------+
                |        bdrv        | button driver
                +--------------------+
                |        SYS:        | SYS interface
  core -> INIT ->|                   | init module
                +--------------------+
                |       BUTTON:      | BUTTON interface
 core <- CLICK <-|       @i,n        | button @i n x clicked
  core <- HOLD <-|       @i,ms       | button @i held over ms
                +--------------------+
```

Figure 17*: module interface* of driver module `bdrv`

Condensed Module Interfaces

One of our goals is to provide high-level descriptions of event driven applications. The term *high-level* suggests focusing on the essentials and omitting details which are

less relevant. This principle can be applied to *module interfaces* by condensing each set of *interactions* belonging to the same *interaction class* to the related *interaction class*. This leads us to the concept of *condensed module interfaces*, which are displayed in figure 18 for our toy application.

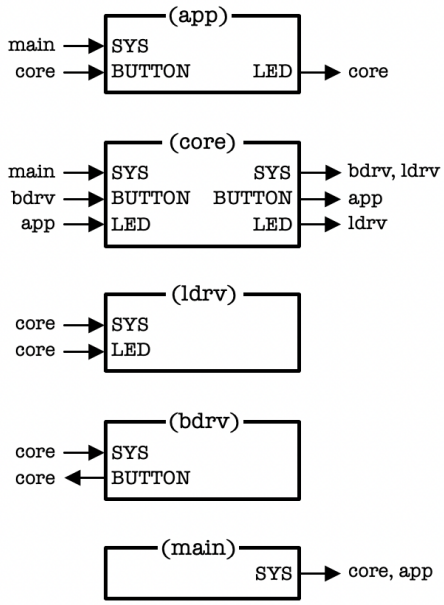

Figure 18*: condensed module interfaces* for toy application

Interaction Graph

Based on the concept of *condensed module interfaces* we can further introduce the concept of an *interaction graph*, which describes which module interacts with some other module regarding a specific *interaction class*.

```
main >> SYS >> app;
core >> BUTTON >> app;
app >> LED >> core;
main >> SYS >> core;
bdrv >> BUTTON >> core;
core >> LED >> ldrv;
core >> SYS >> ldrv;
core >> SYS >> bdrv;
```

Figure 19: Text form of toy application's *interaction graph*

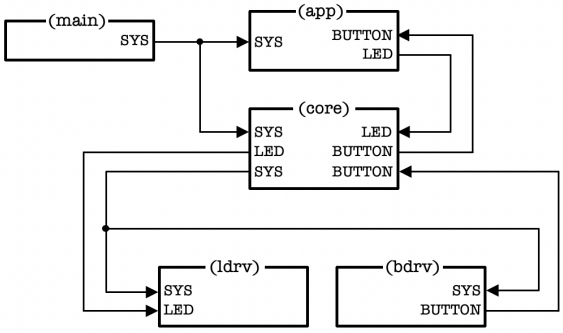

Figure 20*:* graphical form of toy application's *interaction graph*

The text representation of the *interaction graph* has also a graphical representation (figure 20). Each line of the *interaction graph's* text representation has its equivalent arrow from the source module to the destination module in the graphical representation.

In our experience the graphical representation of the *interaction graph* is a powerful form of describing module interactions on a high level.

- In the *interaction graph* all kinds of *module interactions* are grouped in terms of *interaction classes.*
- It allows us to draw a complete picture of an application's *interaction topology* in condensed form.
- Alternatively, when considering *interaction subgraphs,* we have a powerful tool to emphasize some main aspects of an event driven application (see figure 21, where all initializing interactions have been dropped)
- The reader may realize that all information for the construction of the *interaction graph* or of any *interaction subgraph* is also contained in the *interaction profile*.

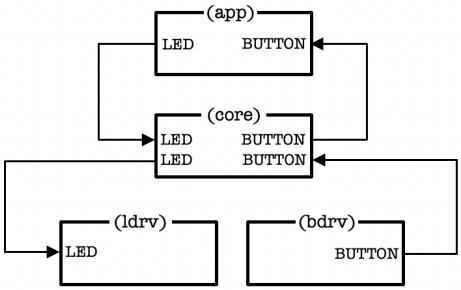

Figure 21: *interaction subgraph* related to `Toggle_Sequence` and `Off_Sequence`

Initializing Graph

Experience tells us that the order of module initialization is essential. For this reason, let us investigate the *interaction subgraph* related to the `Init_Sequence` (figure 22), which we call the *initializing graph*.

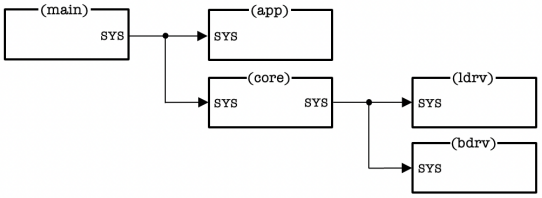

Figure 22: Graphical form of toy application's *initializing graph*

While we have in general circular dependencies in *interaction graphs* (e.g.: `core >> BUTTON >> app >> LED >> core`) the *initializing graph* has a *tree structure*, or in other words, it is strictly hierarchical organized. Hierarchical structures enjoy some preference in software architectures as they define clear responsibilities for subsystems. In the current case the `main` module is respon-

sible for initializing the high-level layer (represented by `app`) and the low-level layer (represented by `core`). Module `core` is responsible for the initialization of the drivers (`ldrv` and `bdrv`).

System Graph

In our toy application *interaction class* `SYS` comprises only one *interaction* `SYS.INIT` (the initializing interaction). In general, the `SYS` class could consist of many more *interactions*. Let us claim the following requirement:

- No matter how many interactions the `SYS` class comprises, the *interaction subgraph* related to the `SYS` class has always a tree structure (i.e., does not have cyclical dependencies).

Assuming such requirement being satisfied, let us do the exercise to replace all *interaction class tags,* which are not equal to `SYS`, by *class tag* `OTHER`. The *interaction graph* of figure 19 and 20 becomes then the reshaped form of figure 23 and 24:

```
main >> SYS >> app;       // down
main >> SYS >> core;      // down
core >> OTHER >> app;     // up
core >> SYS >> ldrv;      // down
core >> SYS >> bdrv;      // down
app >> OTHER >> core;     // down => drop
bdrv >> OTHER >> core;    // up
core >> OTHER >> ldrv;    // down => drop
```

Figure 23: text form of toy application's reshaped *interaction graph*

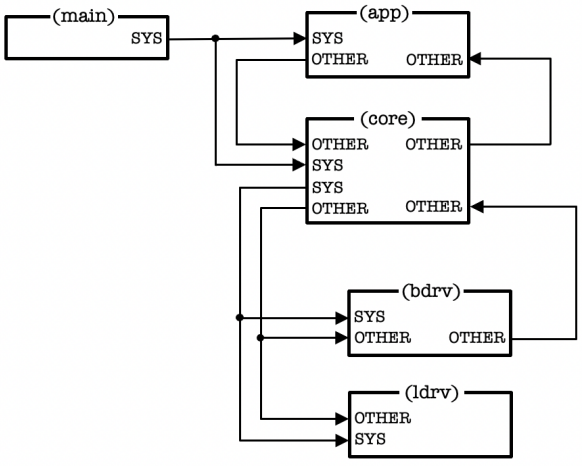

Figure 24: graphical form of toy application's reshaped *interaction graph*

According to the tree structure defined by the SYS interactions we can identify a direction of each interaction flow, which is either *up, down* or neither *up* nor *down*, which we call *cross* (not occurring in our toy application).

In the next step we drop all `OTHER` related *interactions* which are in the *down* direction. This means that we drop all lines in figure 23 which are commented with "// down => drop", leading us to the so-called *system graph* (figure 25).

```
main >> SYS >> app;       // down
main >> SYS >> core;      // down
core >> OTHER >> app;     // up
core >> SYS >> ldrv;      // down
core >> SYS >> bdrv;      // down
bdrv >> OTHER >> core;    // up
```

Figure 25: text form of toy application's *system graph*

Based on the *system graph* we can draw a very interesting chart, the so-called *system interaction chart* (figure 26).

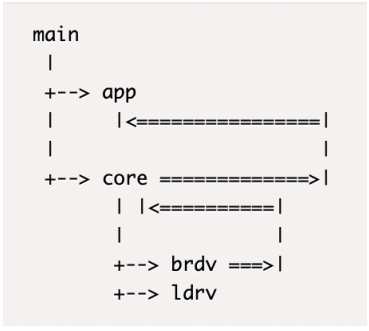

Figure 26: *system interaction chart* of toy application

In figure 26 all *interactions* with *down* direction (e.g., `SYS` interactions) appear as thin arrows (-->), while *interactions* with *up* or cross direction (in our example there are no *cross* interactions) are displayed as thick arrows (==>).

The system interaction chart is a very valuable chart, especially during the design phase of the system architecture, but also as a documentation, for the following reasons:

- The *system interaction chart* has even a higher level of abstraction than the *interaction graph*, and thus gives an even more compact overview of the *module interaction topology*.
- The *system interaction chart* defines initializing responsibilities for subsystems, and in consequence the initializing *interaction sequence*, which is a very important *interaction sequence* in every event driven embedded application.
- Further the *system interaction chart* defines a draft framework for the *interaction graph* of an event driven embedded application, providing a guideline during the design phase of the *interaction profile*.
- The *system interaction chart* clearly defines how interactions should be conveyed *down* or *up* in a layered module hierarchy. The fewer *cross* interactions are involved, the cleaner is the software architecture of the application (i.e., the easier is it understandable and maintainable).

The system interaction chart is a simple chart and can be established with a text editor, and we highly recommend adding this chart in an early phase to every event driven embedded application (`main.c` or `main.h` would be a good location in a C-program). It forces the team to think thoroughly about how *down-stream* or *up-stream interactions* should be conveyed.

Coding

To recap, we started from a given *interaction profile* and recognized that the *interaction profile* is like a DNA of an event driven embedded application, allowing us to extract *interaction classes*, *(module) interactions* and *interaction sequences* with its graphical representations of *interaction sequence charts*. Further it allowed us to extract *module interfaces* in detailed or condensed form, as well as *interaction graphs* in text or graphical form. Finally, we could extract the *initializing graph*, *system graph* and *system interaction chart* from the *interaction profile*. That's a lot!

The reader should realize that all these concepts are not related to any specific programming languages. Anyway, it would be interesting to see now specific code which implements the interaction profile of figure 9. For this reason, let us focus on the following (sample) *interaction*

```
app >> LED.SET(i,on) >> core
```

and find some related C-code which implements this *interaction*. The reader may be noticed that there is more than one way of C-code implementation, and we will start with an intuitive approach related to conventional C-coding.

A Conventional C-Coding Approach

The following approach is demonstrated by sample application `01-interact` [6] comprising two acting modules `app` and `core` according to figure 27. In addition, a `main` module initiates the interaction for demonstration (not shown in figure 27).

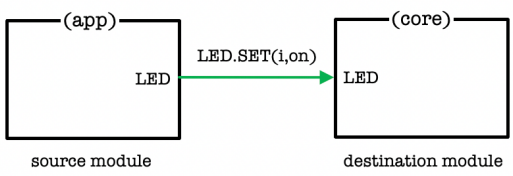

Figure 27: simple interaction

Using conventional C-coding style such interaction would be handled by `core` in terms of a public *interaction handler* function, which we name `core_LED_SET()`. Thus, the sample code for module `core` (header `core.h` and implementation `core.c`) could look as follows:

```
// core.h - hardware core module (01-interact)
#ifndef __CORE_H__
#define __CORE_H__

int core_LED_SET(int i, int on);  // turn LED @i on/off

#endif // __CORE_H__
```

Listing 1: header file core.h (01-interact)

```
// core.c - core module implementation (01-interact)

#include <stdio.h>
#include "core.h"

int core_LED_SET(int i, int on)
{
  printf("core: LED.SET(%d,%d)\n",i,on);
  return 0; // no error
}
```

Listing 2: implementation file core.c (01-interact)

This demonstrates the destination part of the *interaction*. Let us also see the corresponding source part by implementing a sample function `invoke()` in module `app`, which initiates the *interaction*.

```
// app.h - application module (01-interact)
#ifndef __APP_H__
#define __APP_H__

  void invoke(void);  // invoking an interaction

#endif // __APP_H__
```

Listing 3: header file app.h (01-interact)

```
// app.c - implementation of module app (01-interact)

#include "app.h"
#include "core.h"

void invoke(void)      // invoking a module interaction
{
  int i = 1, on = 0;   // sample values
  core_LED_SET(i,on);  // app >> LED.SET(i,n) >> core
}
```

Listing 4: implementation file app.c (01-interact)

To see the sample running we add a `main` module providing function `main()` with the task to call sample function `invoke()`.

```
// main.c - main entry point for sample 01-interact

#include "app.h"

int main(void)  // main entry point
{
  invoke();   // invoking a module interaction
  return 0;     // no error
}
```

Listing 5: main.c – main entry point (01-interact)

This completes the code of `01-interact`. Building and running the code [6] produces the output:

```
core: LED.SET(1,0)
```

Let us summarize:

- Step-by-step translation of an *interaction profile* to C-code by consequent application of the just presented principles to each *interaction* of the *interaction profile* figures out to be a *straight forward task*.

But Wait a Minute!

Try to study the code of `01-interact` by reading only the public interfaces in the header files `app.h` and `core.h` without looking into the implementation files `app.c` and `core.c`. Would we be able to find some indications about which module initiates some module interaction?

The answer is "No"! It means that by using this kind of C-programming style we are left without any idea of where which kind of interactions are initiated, unless we read through the implementation code. For a tiny application like our toy app this is not an issue, but in real applications the implementation code makes 80% to 85% of the total.

Also think about this: In phase 2 of the proposed development process of figure 2 we had the plan to produce some outcome which gives the team a high-level understanding of the module interactions. Yes, it is true that with the development of the *interaction profile* we have moved a good step forward towards our goal. On the other hand, regarding an efficient software development process, we should target that all the information of the interaction profile should also be casted into code.

But since we recognize now that with the conventional style of C-coding some information of the *interaction profile* is casted into *header files*, while some other part of information is casted into *implementation files*, we are facing the issue that the implementation code is not planned to exist at stage 2 of the development process!

- The conclusion is that with conventional C-coding style we are not able to support an efficient software development process according to figure 2.

This is a bold statement! As embedded programmers we've been grown-up with C-programming and took over the C-programming style from other C-programmers. Now, that we are used to this style, we are blind to recognize a fundamental issue of the conventional programming style, since we have learnt how to read through a C-program which has been coded by someone else. And for a tiny application like our toy application this would anyway not be a harm.

But think about it! The style we are all used to in embedded software development is like reading a book, where we start reading the first chapter which confronts us with some new terminology. Instead of explaining new terminology before using it, the author of the book expects us to seek forward in the following chapters without any particular guidance to find the location, where we get the new terminology explained.

Let's do a side jump to electronic hardware design. Look at the hardware schematics of a simple breakout board shown in figure 28.

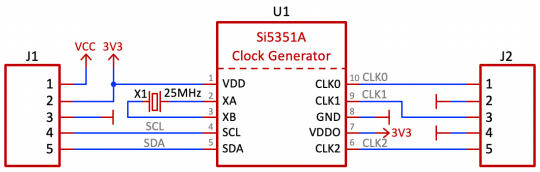

Figure 28: hardware schematics of simple breakout board

How would we analyze such schematics as a hardware developer? It is not difficult to understand what is going on here. There are three modules: `J1`, `J2` are connector blocks, while `U1` is a (clock and timing oscillator) IC (integrated circuit) of type Si5351A, for which detailed documentation can be looked-up on the internet. Finally, there is a crystal `X1`.

The schematics defines precisely the signal flow between different modules. E.g., the `SCL` signal flows from `J1`/pin 4 to `U1`/`SCL`, the `SDA` signal from `J1`/pin 5 to `U1`/`SDA`, and so on. The overall *module interaction topology* is well defined and well recognizable by the schematics of figure 28.

Imagine now that we remove all information about module outputs and interconnections in figure 28, which leads us to figure 29.

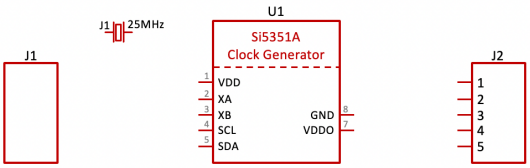

Figure 29: Incomplete Hardware schematics of a simple breakout board

Then we add this kind of removed information chunks to the data sheets of connectors `J1`, `J2` and IC `U1`, and give the total package to a hardware design engineer for investigation and analysis.

What would the hardware engineer answer if we ask him, which of the two "hardware descriptions" are easier to understand? We trust the reader to agree that a schematic of figure 28 looks understandable and complete, while the schematics of figure 29 looks incomplete and rises many questions.

On the other hand, conventional embedded software is written in the style of figure 29, and no embedded software engineer has any trouble with such style.

Pluggable Modules

There are some alternative approaches how to avoid the highlighted issue of previous section. Since conventionally C-coded modules are providing function prototypes for *interaction handlers* which act as *input interfaces* we must seek for a representation of related *output interfaces*. Modules with input and output interfaces could then be plugged together outside of module implementations according to the scheme of figure 30.

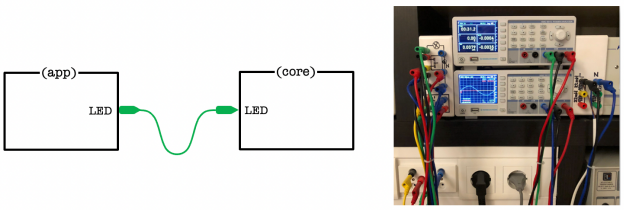

Figure 30: modules which are plugged together

In the simplest way this could be achieved by implementing an *output interface* for interaction `LED.SET` as a public function pointer to an *interaction handler* function. This function pointer would be visible in the public module interface (`app.h`) and would provide the information that module app is initiator of an `LED.SET` *interaction*. In contrast to the situation of figure 29 we would now see all output pins of the modules.

Sample application 02-interact [7] demonstrates the required changes of module `app` (listing 6 and 7), while no change is required for module `core`.

```
// app.h - hardware core module (02-interact)
#ifndef __APP_H__
#define __APP_H__

  void invoke(void);  // invoking a module interaction

  int (*app_LED_SET)(int i, int on); // turn LED @i on/off

#endif // __APP_H__
```

Listing 6: header file `app.h` (02-interact)

```
// app.c - implementation of module app (02-interact)

#include "app.h"

void invoke(void)        // invoking a module interaction
{
  int i = 1, on = 0;     // sample values
  (*app_LED_SET)(i,on);  // app >> LED.SET(i,n) >> ...
}
```

Listing 7: implementation file `app.c` (02-interact)

There is another subtlety in this approach. The reader should notify that the line

```
#include “core.h”
```

has been dropped from `app.c`, which means we eliminated a source code dependency. Getting rid of source code dependencies is a big topic in the design of clean architectures (find more information in [2]). The approach towards clean architectures is to reduce *source code dependencies between modules* by making modules only dependent *from interaction interfaces*. We will cover such coding style in the next section.

There is one thing missing, the definition of the module interconnection, which relates to the wiring in figure 28. We provide function `topology()` to perform such setup of the connection topology, and a good place to define such function is in `main` (listing 8).

```
// main.c - main entry point for sample 02-interact

#include "app.h"
#include "core.h"

void topology(void)
{
  app_LED_SET = core_LED_SET; // connect
}

int main(void)  // main entry point
{
  topology();   // setup interaction topology
  invoke();     // invoke a sample interaction
  return 0;     // no error
}
```

Listing 8: `main` module with topology setup (02-interact)

Let us summarize:

- Sample `02-interact` is coded according to a pluggable module architecture.
- In such architecture modules have no source code dependency between each other.
- Public module interfaces (header files) provide now *input interfaces* as well as *output interfaces* of *interactions*.
- The interconnection topology is defined in `main`. Implementation files (intentionally) contain no information about the destination of a module interaction.

In practice this means: To understand the kind of *interactions* and the *interaction topology* of an application with *pluggable module architecture* it is sufficient to consider the main file and the header files, which is typically only 15-20% of an application code.

Further it may be noted that pluggable module architecture is compliant with the proposed software development process of figure 2.

Interaction Interface Based Architecture

Now that we understand the principle of pluggable module architecture, we propose an enhanced C-coding approach which utilizes interaction interfaces. According to figure 5, 6 and 7 we can define C-coded interaction interfaces (listing 9).

```
// interface.h (interaction interface definitions)
#ifndef __INTERFACE_H__
#define __INTERFACE_H__

//============================================================================
//                  +--------------------+
//                  |        SYS         | SYS interface
//           INIT --|                    | init module
//                  +--------------------+
//============================================================================

  typedef int (*_SYS_INIT_)();         // system init, store callback

  typedef struct
          {
            _SYS_INIT_ INIT;
          } _SYS_;                     // SYS interaction interface

//============================================================================
//                  +--------------------+
//                  |       BUTTON       | BUTTON interface
//          CLICK --|        @i,n        | button @i n x clicked
//           HOLD --|        @i,ms       | button @i held over ms
//                  +--------------------+
//============================================================================

  typedef int (*_BUTTON_CLICK_)(int i, int n); // button @i n x clicked
  typedef int (*_BUTTON_HOLD_)(int i, int ms); // button @i held over ms

  typedef struct
          {
            _BUTTON_CLICK_ CLICK;
            _BUTTON_HOLD_ HOLD;
          } _BUTTON_;                  // BUTTON interaction interface

//============================================================================
//                  +--------------------+
//                  |        LED         | LED interface
//            SET --|       @i,on        | turn LED @i on/off
//            OFF --|                    | turn all LEDs off
//                  +--------------------+
//============================================================================

  typedef int (*_LED_SET_)(int i, int on);     // turn LED @i on/off
  typedef int (*_LED_OFF_)(void);              // turn all LEDs off

  typedef struct
          {
            _LED_SET_ SET;
            _LED_OFF_ OFF;
          } _LED_;                     // LED interaction interface

#endif // __INTERFACE_H__
```

Listing 9*: interaction interface* definition (03-interact)

The used coding scheme requires a data structure for each module supporting an (optional) *input interface structure* (`in`) and an (optional) *output interface* structure (`out`).

```
typedef struct
{
  struct { ... } in;     // input interface structure
  struct { ... } out;    // output interface structure
} ...;
```

For module `app`, which needs to implement the `LED` *interaction interface* (type `_LED_`) we find the definition of module type `App` in listing 10. Note that an *output interaction interface* is implemented in terms of a pointer, pointing to an *interaction interface*, which is set up during execution of the `topology()` function.

```
// app.h - hardware core module (03-interact)
#ifndef __APP_H__
#define __APP_H__

#include "interface.h"

typedef struct
{
  struct { _LED_ *LED; } out;
} App;

extern App app;
void invoke(void);  // invoking a module interaction

#endif // __APP_H__
```

Listing 10*: definition of module type* `App` (03-interact)

```
// app.c - implementation of module app (03-interact)

#include "app.h"

void invoke(void)         // invoking a module interaction
{
  int i = 1, on = 0;      // sample values
  app.out.LED->SET(i,on); // app >> LED.SET(i,n) >> ...
}

App app; // module instance
```

Listing 11*: declaration of module instance* `app` (03-interact)

```
// core.h - hardware core module (03-interact)
#ifndef __CORE_H__
#define __CORE_H__

#include "interface.h"

typedef struct
{
  struct { _LED_ LED; } in;
} Core;

extern Core core;

#endif // __CORE_H__
```

Listing 12*: definition of module type* `Core` (03-interact)

```
// core.c - core module implementation (03-interact)

#include <stdio.h>
#include "core.h"

static int LED_SET(int i, int on)
{
  printf("core: LED.SET(%d,%d)\n",i,on);
  return 0; // no error
}

Core core = {.in={.LED={.SET=LED_SET}}};  // module instance
```

Listing 13*: declaration of module instance* `core` (03-interact)

For the `core` module a module type `Core` with input interaction interface `_LED_` needs to be defined (listing 12), which is used for declaring a `Core` instance `core` to be initialized with the address of the local *interaction hand-*

ler function `LED_SET` (see listing 13). The goal is to have a setup as shown in figure 31.

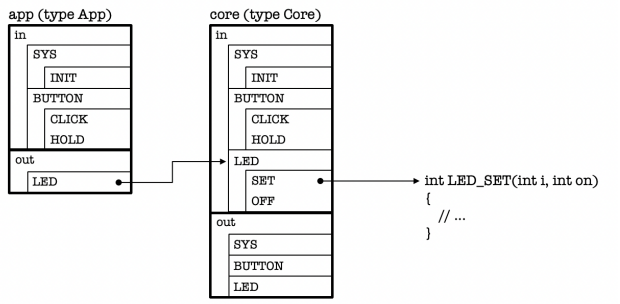

Figure 31: connected modules `app` and `core`

Interaction interfaces are now provided for both *input* and *output* direction. The connection in figure 31 (only one connection is shown) is now explicitly set up in the `topology()` function of main (listing 14).

```
// main.c - main entry point for sample 03-interact

#include "app.h"
#include "core.h"

void topology(void)
{
  app.out.LED = &core.in.LED; // connect
}

int main(void)  // main entry point
{
  topology();   // setup interaction topology
  invoke();     // invoke a sample interaction
  return 0;     // no error
}
```

Listing 14: `main` module with topology setup (03-interact)

It should be highlighted that the connection in function `topology()` is established for all *interactions* belonging to the `LED` class. This is comparable to a hardware schematics like figure 28, where the wire lines `SCL` and `SDA` are replaced by a bundle connection as shown in figure 32.

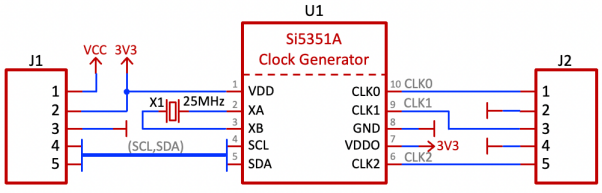

Figure 32: hardware schematics with bundle connection

For the complete C-code of our toy application the interested reader is referred to [9].

Let us summarize:

- There are different kinds of C-code implementation for an interaction profile
- All header files and the skeleton part of the implementation files can be automatically generated.

Summary

- We emphasized to use a strictly modular *layered architecture* for event driven embedded applications.
- We introduced the fundamental concept of *interactions* to represent any kind of module interactions.
- The *interaction profile*, a composition of the *module list* and all *interaction sequences*, is a powerful high-level description of an event driven application.
- An *interaction profile* is like a DNA of an application. It allows to extract *interaction sequences*, *interaction interfaces*, *interaction sequence charts*, *module interfaces*, the *interaction graph*, *the initializing graph*, the *system graph* and the *system interaction chart*.
- None of the presented concepts are tied to a specific programming language
- The *system interaction chart* describes the *interaction topology* in a very condensed form. It is a valuable tool for the design phase of the software architecture, and it should not be missing in a software documentation.
- Different approaches have been demonstrated to translate an *interaction profile* into C-code, where benefits of pluggable module architectures have been highlighted.
- Finally, the *interaction profile* allows automatic C-code generation of all *public module interfaces* (defined in header files) and allows also automatic skeleton C-code generation for the *implementation files*.
- Pluggable software architecture allows to extract the *module interaction topology* from the main file and header files, without consultation of any implementation file.

References

[1] Robert C. Martin: Clean Code - *A Handbook of Agile Software Craftsmanship*; Adison Wesley 2009

[2] Robert C. Martin, J. Grenning, S. Brown: *Clean Architecture - A Craftman's Guide to Software Structure and Design*; Prentice Hall 2018

[3] Zephyr Documentation: *nRF52 DK*; https://docs.zephyrproject.org/latest/boards/arm/nrf52dk_nrf52832/doc/index.html

[4] Adrian Ostrowski, Piotr Gaczkowski: *Software Architecture with C++*; Packt Publishing Birmingham / Mumbai 2021

[5] Benedikt Bollig: *Formal Models of Communicating Systems: Languages, Automata, and Monadic Second-Order Logic*; Springer 2006

[6] Sample code `01-interact`: download link https://github.com/bluccino/principles.git

[7] Sample code `02-interact`: download link https://github.com/bluccino/principles.git

[8] Sample code `03-interact`: download link https://github.com/bluccino/principles.git

[9] Sample code `04-toy-app`: download link https://github.com/bluccino/principles.git