Deadlock/reader-sw

Release 2.0 alpha

May 03, 2018

Contents

Wow, such empty

CHAPTER 1

Table of contents

Reader Firmware Architecture

The Reader uses STM32 microcontroller. To keep the cost of hardware as low as possible, these MCUs are not capable of running mainstream OS such as Linux. Since the Reader must be fast and stable, the firmware must be written in a way that it can execute efficiently on an MCU - therefore in C.

Writing a firmware in C for bare-metal, however, has several disadvantages. One big disadvantage is portability: should the Reader require hardware change the firmware would have to be laboriously ported as well, including possible rewrite of bare-metal drivers. Another disadvantage is high maintenance cost and entry barrier for new developers entering the project.

To mitigate some of these disadvantages, we have decide to base the firmware on ChibiOS. ChibiOS is a free development environment for embedded applications. It contains a RTOS, which provides threading capabilities, synchronization primitives and other useful programming constructs, and HAL, which simplifies development and facilitates firmware porting to different MCUs.

There are multiple ways to write a firmware for embedded devices, from a simple loop to extremely modular but complex microkernel-like designs. Each design method provides a different mix of implementation cost, maintainability, stability and extensibility.

Components and their interaction

Firmware of the Reader will be composed of several Tasks and one Master Task. Each task will run in its separate thread and will do only one thing, for example there will be a task for reading the RFID card or task for updating the UI. These tasks won't communicate directly with each other. They will only communicate with the Master Task. The Master Task will implement business logic of the reader.

Since Tasks and the Master Task run in different threads, some form of inter-thread communication will have to be set up. In general, tasks will provide a thread-safe API which the Master Task may call whenever it needs. This thread-safe API will internally, in a task-specific way, transport the message to the task thread. This is done so that each task may choose communication method best suited for it. For example, RFID Card Task may support only 2 commands: Start Polling and Stop Polling. The simplest way to implement these commands is to just set a bit in shared memory, update of which is always atomic. On the other hand, the Communicator Task will have to receive a data structure describing message the Master Task wishes to send. The Master Task may even request sending a message if another message is being sent just now. Most fitting communication mechanism for this particular task is the Mailbox construct.

Watchdog

The multithreaded nature of this firmware also brings some issues. Since each task runs as a separate thread, it may lock up / enter an endless loop without impairing the rest of the system. For example, RFID Card Task may enter an endless loop. Since other threads, including the Master Task may run with higher priority, they won't notice anything abnormal, except that when user presents a card to the reader, the card won't be detected. Therefore it is necessary to implement a watchdog which would watch over all tasks (including the Master Task), and if something like this happens, it would reset the Reader.

At this point one may be tempted to come up with a complex scheme where each task can be restarted without affecting any other tasks. Indeed, the first working draft of the firmware design included this capability. Implementation, however, proved to be quite challenging, and would require implementation of some sort of garbage-collecting mechanism. In the end, we have decided that added implementation complexity and potential bugs were not worth the advantages such mechanism would bring. Indeed, the whole Reader firmware is able to fully boot up in a fraction of a second after reset, so it is easier to just go for a complete restart.

The implemented watchdog mechanism utilizes the hardware watchdog built in the MCU. The watchdog in this MCU is essentially just a timer, which when runs out, resets the MCU. So the firmware has to perpetually reset this timer. Resetting the watchdog is the responsibility of the Master Task.

Each task, including a Master Task, should generate Heartbeats. This Heartbeat should depend on correct operation of the given Task. For example, the RFID Card Task tries to read a card in a loop. After each loop, it can generate a Heartbeat. If the task gets stuck when reading a card, it won't be able to generate a Heartbeat.

The Master Task internally keeps a vector of bits, where each bit represents a task (Heartbeat Vector). Each task sends a heartbeat message to the Master Task each time the task generates a Heartbeat. When the Master Task receives that message, it sets a bit in the Heartbeat Vector. Then, when the Master Task itself generates a Heartbeat, it checks the Heartbeat Vector, and if all bits are set, it resets the Watchdog, and clears all bits in the Heartbeat Vector. Therefore if some tasks continually misses its Heartbeats, its bit won't ever be set and the Watchdog won't be reset.

Detailed descriptions

Task

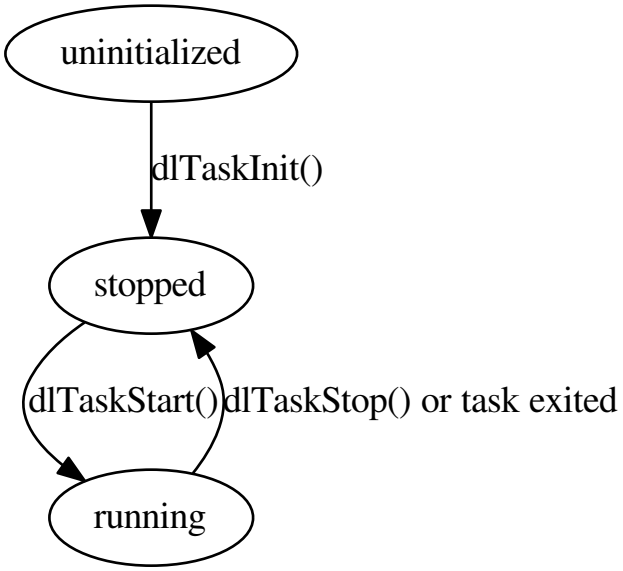
The Deadlock Reader firmware is composed of several Tasks and a signle Master Task. Each task will run in its separate thread and will do only one thing, for example there will be a task for reading the RFID card or task for updating the UI. These tasks won't communicate directly with each other. They will only communicate with the Master Task. The Master Task will implement business logic of the reader.

Tasks are located in subfolders `src/tasks`, one folder per task. Each task has a header, which defines an interface for that task.

To lower implementation and maintenance costs, there can only be a single instance of each task. Task interface functions therefore operate on a global instance of the given task.

Task states

The following state machine represents an abstract task:



- `uninitialized`: internal structures of the task are not initialized. The task can't be started and no task API functions are callable. The task won't call Master Task callbacks.
- `stopped`: internal task structures are now initialized and the task can be stared. No task APIs can be called and the task won't call Master Task callbacks.
- `running`: the task is functioning normally (the task thread is running). Task API is available and the task will call Master Task callbacks. In particular, each task is required to periodically call Heartbeat callback.

All Tasks implement this state machine. The Master Task is responsible for initializing, starting (and if required stopping) Tasks.

The Master Task also implements this state machine. The Master Task is initialized and started by the firmware initialization code. This code just starts the task, it does not monitor it and it won't restart it if the task transitions back to `stopped` state.

Task API and naming conventions

Naming conventions of the API are similar to ChibiOS naming conventions: http://chibios.sourceforge.net/docs3/rt/concepts.html#naming

The API functions are following this convention: `dl<group><subgroup><action>()`

- `dl`: stands for Deadlock
- `group`: for tasks API, the Group is always `Task`
- `subgroup`: subgroup is the name of the task
- `action`: what the API function does (`Start`, `Stop`, ...)

Examples: `dlTaskUIStart()` or `dlTaskRFIDStopPolling()`.

Each task must provide functions used in the state machine transitions:

- `dlTask<TaskName>Init()` - initializes task data structures and prepares it to be started
- `dlTask<TaskName>Start()` - starts the task thread
- `dlTask<TaskName>Stop()` - stops the task thread

User Interface Task

User Interface control task.

This task controls the user interface of the Reader. User Interface of the Reader RevA board consists of 2 bi-color (red/green) LEDs, labelled Status LED and Lock LED and of a small speaker. This UI can use a combination of flashes and beeps to inform the user of a certain state or of an event.

The UI has 2 ways of informing the user of something: persistent states and message flashes. The persistent state informs user of some long-lasting condition (like system OK, door locked). The persistent state stays the same and displayed until it is explicitly changed. The message flashes are used to inform the user of one-time events that just happened (card rejected). They will execute a scripted sequence and then automatically return to previous persistent state.

Example: The system starts the UI task, which switches to the default state (Error). The system initializes a connection with the Controller and switches the task to "Normal - Locked" state. A user attempts to open the door using an invalid card. The persistent state stays "Normal - Locked" and "Card Rejected" UI flash will play. The user uses a correct card. The persistent state changes to "Normal - Unlocked" and on top of that "Card Accepted" UI flash will play.

Note Deciding when to unlock and lock the door again is responsibility of the Controller, so even though the door is usually unlocked only temporarily, "Normal - Unlocked" is a persistent state from the Reader's point of view.

Enums

enum `dl_task_ui_state`
User interface states.

These are persistent states which the user interface may be presenting to the User. User is in a given state until it is explicitly changed.

Values:

`DL_TASK_UI_STATE_ERROR`
Error state. Status LED is blinking red. This is the default

`DL_TASK_UI_STATE_LOCKED`
Normal locked. Status LED is green, Lock led is red

`DL_TASK_UI_STATE_UNLOCKED`
Normal unlocked. Status LED is green, lock led is green

enum `dl_task_ui_flash`
User interface flashes.

These are temporary user interface states. They are usually a sequence of actions (a beep, LED blink) displayed on top of the persistent state until the sequence finishes.

Example: UI state is DL_TASK_UI_STATE_UNLOCKED, and flash DL_TASK_UI_FLASH_READ_OK is invoked. This flash will beep once with a high tone and then stop. UI stays in the DL_TASK_UI_STATE_UNLOCKED.

Values:

DL_TASK_UI_FLASH_READ_OK
Card read and auth OK: One long high-pitched beep

DL_TASK_UI_FLASH_READ_BAD
Card read and auth failed: Three short low-pitched beeps

DL_TASK_UI_FLASH_VADER
Vader

Functions

void **dlTaskUiInit**(uint8_t *task_id*, *dl_task_ui_callbacks* **callbacks*)
Task initializer.

This function initalizes internal data structure of the task and sets up callbacks to the Master Task

Parameters

- `task_id`: Our identificator the Master Task has chosen
- `callbacks`: Structure of function pointers to callbacks this task should use

void **dlTaskUiStart**(void)
Task starter.

This function starts the task thread.

void **dlTaskUiStop**(void)
Task stopper.

This function stops the task thread.

void **dlTaskUiSetUIState**(*dl_task_ui_state state*)
Sets the persistent UI state.

Note Thread safety: This function can be called from any thread when the ChibiOS is in Normal state.

Parameters

- `dl_task_ui_state`: UI state to set

void **dlTaskUiFlashMessage**(*dl_task_ui_flash flash*)
Flashes a temporary user state.

Note Thread safety: This function can be called from any thread when the ChibiOS is in Normal state.

Parameters

- `dl_task_ui_flash`: UI flash

struct dl_task_ui_callbacks
#include <ui-task.h> A structure of Master Task callbacks.

User Interface of the Reader has no input elements, so this task does not need to report anything else than heartbeat to the Master Task.

Note These callbacks must be thread safe

Public Members

void (***heartbeat**) (uint8_t task_id)
A heartbeat callback.

A Heartbeat callback of the Master Task. See firmware documentation, section "Reader Firmware Architecture", subsection "Watchdog"

Parameters

- `task_id`: ID of this task, assigned to us by the Master Task

CardID Task

Card ID reading task.

This task uses on-board RFID reader module to try and read IDs of all cards present in the RF field. The underlying library is able to read IDs of all cards at once (if those cards behave properly).

The master task must explicitly request that this task starts polling for cards. When this task is polling for cards it may read one or more card IDs in one poll cycle. When that happens, this task will invoke a callback to the Master Task and will stop polling for cards until the Master Task reenables polling.

When this task is not polling for cards the RFID reader module is in low-power mode.

Functions

void **dlTaskCardIDInit** (uint8_t *task_id*, *dl_task_cardid_callbacks* **callbacks*)
Task initializer.

This function initalizes internal data structure of the task and sets up callbacks to the Master Task

Parameters

- `task_id`: Our identificator the Master Task has chosen
- `callbacks`: Structure of function pointers to callbacks this task should use

void **dlTaskCardIDStart** (void)
Task starter.

This function starts the task thread.

void **dlTaskCardIDStop** (void)
Task stopper.

This function stops the task thread.

void **dlTaskCardIDStartPolling** (void)
Requests that this task starts polling for cards.

Note Thread safety: This function can be called from any thread when the ChibiOS is in Normal state.

void **dlTaskCardIDStopPolling** (void)
Requests that this task stops polling for cards.

After this function is invoked, the poll that is already in progress is finished, however result of that poll will be discarded. It is guaranteed that after this function returns, callback `card_detected` won't be invoked until

`dlTaskCardIDStartPolling` is called. If this function is called during the `card_detected` callback invocation it will block until the callback returns.

Note Thread safety: This function can be called from any thread when the ChibiOS is in Normal state.

struct `dl_task_cardid_callbacks`
#include <cardid-task.h> A structure of Master Task callbacks.

Note These callbacks must be thread safe

Public Members

void (***heartbeat**) (uint8_t task_id)
A heartbeat callback.

A Heartbeat callback of the Master Task. See firmware documentation, section "Reader Firmware Architecture", subsection "Watchdog"

Parameters

- `task_id`: ID of this task, assigned to us by the Master Task

void (***card_detected**) (dl_picc_uid *cards, uint8_t len)
Card Detected callback.

This callbacks informs the Master Task that one or more cards were detected in the RF field and sends their IDs. After this callback is invoked the task will stop polling for cards and won't change contents of parameter `cards`. However, when the Master Task requests that this task should start polling for cards again, data pointed to by `cards` may change at any moment!

Parameters

- `cards`: An array of IDs of detected cards
- `len`: Length of the array of detected cards

void (***reader_error**) (void)
RFID Reader Module error.

The reader module has experienced an unrecoverable error and can't function. The task will automatically stop polling for cards.

Comm Task

Communication handling task.

This task handles serial port communication with Controller.

An intent to send something is delivered as ChibiOS Message. This message is then serialized to CBOR format according to `dcrcp` (DeadCom Reader<->Controller Protocol) schema. The resulting byte buffer is then packed to `dcl2` (DeadCom Layer 2) frame and transmitted over RS232 link.

This task handles sending and receiving `dcrcp` messages, and `dcl2` link management.

Enums

enum dl_task_comm_linkstate
Values:

DL_TASK_COMM_LINKUP

DL_TASK_COMM_LINKDOWN

Functions

void **dlTaskCommInit**(uint8_t *ctrl_task_id*, uint8_t *rcv_task_id*, *dl_task_comm_callbacks* **callbacks*)
Task initializer.

This function initalizes internal data structure of the task and sets up callbacks to the Master Task

Parameters

- `task_id`: Our identificator the Master Task has chosen
- `callbacks`: Structure of function pointers to callbacks this task should use

void **dlTaskCommStart**(void)
Task starter.

This function starts the task thread.

void **dlTaskCommStop**(void)
Task stopper.

This function stops the task thread.

void **dlTaskCommSendSysQueryResp**(uint16_t *rdrClass*, uint16_t *hwModel*, uint16_t *hwRev*, char *serial*[DCRCP_SERIAL_MAX_LEN], uint8_t *swVerMajor*, uint8_t *swVerMinor*)
Send System Query Response CRPM.

void **dlTaskCommSendRdrFailure**(char **str*)
Send Reader Failure CRPM.

void **dlTaskCommSendAM0GotUids**(dl_picc_uid **uids*, size_t *uids_len*)
Send Auth method: PICC UID obtained CRPM.

struct dl_task_comm_callbacks
#include <*comm-task.h*> A structure of Master Task callbacks.

Note These callbacks must be thread safe

Public Members

void (***heartbeat**)(uint8_t task_id)
A heartbeat callback.

A Heartbeat callback of the Master Task. See firmware documentation, section "Reader Firmware Architecture", subsection "Watchdog"

Parameters

- `task_id`: ID of this task, assigned to us by the Master Task

void (*`linkChange`) (*dl_task_comm_linkstate* new_link_state)
A link status has changes.

Either the link was established dropped

void (*`rcvdSystemQueryRequest`) (void)
A System Query Request CRPM has been received.

void (*`rcvdActivateAuthMethods`) (DeadcomCRPMAuthMethod *methods, size_t methods_len)
An Activate Auth Methods CRPM was received.

void (*`rcvdUiUpdate`) (DeadcomCRPMUIClass0States uistate)
UI Update CRPM was received.

Automated testing of the firmware

We want to test our code. However, testing code for embedded devices is not easy since in production environment the code runs on an embedded hardware. There are basically 3 options:

- Compile the code for PC and run tests locally
- Run tests on an emulator of given embedded platform
- Run tests on physical embedded hardware

Some of them have advantages, all of them have disadvantages.

Compiling code for the PC and running the tests locally is very fast compared to other options. However, only hardware-independent pieces of code can be tested in this way. Moreover, since different compiler is used to compile the unit tests these tests won't catch compiler-introduced bugs (which is not unheard of in embedded development). However, logical error is still a logical error no matter the platform or compiler, so these tests can test business-logic well. Although it is possible to test higher-layers of device drivers by mocking the comm interface and emulating the hardware, it is pointless since it is way harder to write a flawless hardware emulator (bug-for-bug compatible with the real hardware) than it is to write a solid driver for it.

Running code on an emulator has an advantage of using the same compiler for both tests and production code. Also the registers may be changed at will and error conditions introduced. Unfortunately, we have not found suitable emulator for our platform.

Running tests on physical embedded device is the most difficult approach. Physical hardware can't be forced to deterministically create a specific failure mode through software when debugging drivers. Collecting test results is also difficult. Unfortunatelly, this is often the only applicable choice.

Chosen testing strategy

We want to do 3 types of testing: unit testing, integration testing and system testing.

Let's start from the lowest layer: device drivers. They will not have automated tests for reasons described above. We won't write many drivers, just one or two, since we are using ChibiOS HAL. They will be tested manually as much as possible though.

The firmware has modular design. Each module runs in its own thread and communicates with other modules using mailboxes. Therefore each module can be separated from other modules and tested on its own. This way we can test how well units integrate into modules and with ChibiOS. These integration tests will run on physical embedded hardware, since ChibiOS must run to run the thread, handle the mailboxes, etc.

Modules can be broken down to units, and these will be unit tested. Unit tests will run on a PC so that they can be written, debugged and run easily.

System testing will be entirely different. Production firmware will run on a production hardware and the whole reader will be treated as a black box. Interaction with other components over external interfaces will be tested using a bed of nails.

Unit tests

Unit tests are built on the Unity unit testing framework and is using the Fake Function Framework.

Unit tests are stored in the `test/` folder. This folder contains subfolders `src/` and `hal/`. Overally, folder structure in the `test/` folder is the same as the folder structure of the sources. For each source file there may be several test files (with the same name as the file under test with suffix `--testNUM.c`).

Technicalities of unit-testing C code with mocking

If one `.c` file represents one unit then unit-testing is relatively easy: compile the file under test, compile the test file, let the test file define mock functions, link them. Test file will call a function from the file under test, it will call some library function which is provided by the mock in the test file.

However, usually a single `.c` file is a single module whereas a function is an unit. Functions in a `.c` file may use other functions from that file, so in order to write an unit test for given function from the `.c` file, other functions from this `.c` file which are called by the function under test may need to be mocked. Even worse, these functions may be static (due to optimalizations). There are several solutions to this problem:

- Split the `.c` file into multiple files so that each file is an unit. Advantage is that building and writing tests is easy. Also it may encourage better and more modular design. Disadvantages are that you need to produce more `.c` files and the code may be harder to write. Also static functions can't be used.
- You can use (through some preprocessor magic like custom defined `testable` macro or by `objcopy --weaken`) weak references. You can then provide your own implementation of custom symbols in the test file and linker will replace weak references in the file under test. Advantage is that writing tests is easy, disadvantage is that you may need magic macros or a bit more complicated build process. You still need preprocessor magic to solve `static` functions, as compiler may do what it wants with `static` functions (such as inline them or change their calling convention).
- You can manipulate (rape?) the GOT table at runtime and force the running program to use your function instead. Building tests is easy, writing them is hard and the whole process is messy. Still won't solve the problem with `static` functions.

We will do the following: where possible and logical we will split the code to multiple `.c` files. It is quite possible that the split will not be needed in most cases, and when it is needed the file should anyway be logically splitted. Additionally the file will be processed with `objcopy --weaken`, so if it **really** is not logical to split the file under test it is still possible to mock the internal functions. Problem with `static` functions will be solved using `testable_static` macro (where it is warranted), which resolves to `static` when compiled normally, otherwise it will resolve to nothing.

Used mocking framework

We are using Unity unit testing framework. This framework is also used by CMock mocking framework, from the same creators, so it would be logical to use it.

That, however, turned out to be a bit problematic.

The CMock is able to parse a header file and produce a mock file for functions it finds. That automatically implies that CMock treats one file as one unit and makes it impossible to mock functions inside that file.

The other problem is the build system. The mocks should be generated automatically and tests should be build automatically as well. Officially recommended option is to use Ceedling as a build system. However, this is not applicable to this project. This firmware is based on ChibiOS, build system of which is based on Makefiles including other ChibiOS-specific makefiles from within ChibiOS folder structure. CMock supports Makefiles, however it is quite inconvenient to use it. It works by generating a new Makefile using a Ruby script. This script makes assumptions about the project which don't hold true for this firmware (e.g. flat source code structure). Moreover, it was required to run `make` twice for this to work. The last option is to use ruby build system `rake`. This was actually feasible, since you are free to write what you want, and the example was easily modified to build the tests with our code structure. Rake was invoked from the Makefile with proper enviroment variables (because they are known only during `make` execution). However, this solution is not elegant since it mixes two different build systems.

All that would be still quite acceptable. However, the biggest problem is that CMock doesn't work with ChibiOS. This is a somewhat innacurate statement, I'll explain. CMock is able to parse a header file and find function definitions within. However, it is not able (nor it should be able) to preprocess that file. In case of ChibiOS one includes only `ch.h`. Other parts of the system are included by this header. Of course, CMock won't look for them and will only see file without function declarations. Workaround around this was to include a real ChibiOS header file, however it was somewhat inelegant, and not universally applicable. Due to preprocessor magic the resulting set of included header files is dependent on compile-time definitions. This makes perfect sense for embedded projects, as it minimizes footprint of the compiled code. However, it also means that in order to reliably generate mocks the files have to be preprocessed. Official advice on this was to run the header through the C preprocessor. So I've done that, and then CMock choked itself with a parsing error on the resulting file.

That's when I've decided that although CMock is a decent framework it is not applicable for my use case.

I've decided to use the Fake Function Framework instead. It's usage: `#include "fff.h"`. Done. Works. I have to do mocks manually, however, i **can** do mocks manually. Usually I don't have to mock that many functions anyway, and all it takes is `FAKE_VOID_FUNC(halInit);`. And if I forget to mock something the linker kicks me in the balls. Bonus: I can use the weak-reference trick and mock functions from the same file that the function under test comes from.

Building and running tests

Unity includes an example Makefile which can be used to build and run unit tests. It needs to be modified to be usable in our case, because:

- It presumes a flat source / test folder structure
- It doesn't automatically generate test runners
- It doesn't automatically weaken references of object file under test
- It supports only one test for each file

So, this will be explanation very similar to the official Unity Makefile explanation, however modified for our purposes.

Let's start by finding all test source files in the test folder with suffix `-testNUM.c`.

```
TEST_CSRC = $(shell find $(TEST_PATH) -type f -regextype sed -regex '.*-test[0-9]*\.c
↪')
```

For each of this test file we will want to create a `.result` file which will store the results. Folder structure in any of the temp folders (either `results` folder or any other test build folder) will be the same as the one in the `test/` directory.

```
RESULTS = $(patsubst $(TEST_PATH)%.c,$(TEST_RESULTS)%.result,$(TEST_CSRC))
```

Now let's begin from the end. At the and we will want to print the test statistics by reading all `.results` files.

```
TEST_EXECS = $(patsubst $(TEST_RESULTS)%.result,$(TEST_BUILD)%.out,$(RESULTS))

run-tests: $(TEST_BUILD_PATHS) $(TEST_EXECS) $(RESULTS)
    @echo
    @echo "----- SUMMARY -----"
    @echo "PASS:   `for i in $(RESULTS); do grep -s :PASS $$i; done | wc -l`"
    @echo "IGNORE: `for i in $(RESULTS); do grep -s :IGNORE $$i; done | wc -l`"
    @echo "FAIL:   `for i in $(RESULTS); do grep -s :FAIL $$i; done | wc -l`"
    @echo
    @echo "DONE"
    @if [ "`for i in $(RESULTS); do grep -s FAIL $$i; done | wc -l`" != 0 ]; then \
    exit 1; \
    fi
```

This recipe of course depends on all result files. Other dependencies are a bit hacky:

- `$(TEST_BUILD_PATHS)`: folders temporarily build files live in
- `$(TEST_EXECS)`: test executables. Technically this should not be here, since each `.result` file should depend on a single executable file. However, this causes executable files to be build in bulk before the tests are run and that makes the test output much more readable and less cluttered.

Each `.result` file is created by executing a single test:

```
$(TEST_RESULTS)%.result: $(TEST_BUILD)%.out
    @echo
    @echo '----- Running test $<:'
    @mkdir -p `dirname $@`
    @-./$< > $@ 2>&1
    @cat $@
```

Each executable test is created by linking the following:

- Test runner (contains the entry point and executes tests)
- Unity framework
- Test file
- File under test

```
.SECONDEXPANSION:
FILE_UNDER_TEST := sed -e 's|$(TEST_BUILD)\(.*\)-test[0-9]*\.out|$(TEST_OBJS)\1-under_
↪test.o|'
$(TEST_BUILD)%.out: $$(shell echo $$@ | $$(FILE_UNDER_TEST)) $(TEST_OBJS)%.o $(TEST_
↪OBJS)unity.o $(TEST_OBJS)%-runner.o
    @echo 'Linking test $@'
    @mkdir -p `dirname $@`
    @$(TEST_LD) -o $@ $^
```

Now there is a bit of Makefile hackery going on there. This recipe generates an executable which runs tests, named for example `build/tests/out/src/main-test12.out`. It does that by linking `build/test/objs/unity.o`, `build/test/objs/src/main-test12.o` and `build/test/objs/src/main-test12-runner.o`. It also needs `build/test/objs/src/main-under_test.o`, which is the file under test.

The question is how to get the name of the file under test. If there was only one test file per source file it would be easy, `%` could be used as with other things. However, there are more test files per source file. Name of the test file must match a certain pattern (regex `\(.*\)-test[0-9]*\.out`), and this pattern needs to be re-

placed. This is not possible with make's `%`. That's why we use the `sed` command, which makes for example `build/test/objs/src/main-under_test.o` from `build/tests/out/src/main-test12.out`.

The last problem is how to run `%` or `$@` through the shell with sed. Makefiles are read in two phases: in the first phase make reads in the Makefile and includes, expands so-called `immediate` variables and constructs a dependency graph. In the second phase make determines what needs to be rebuilt and rebuilds it. The problem is that recipe name and dependencies are immediate variables, therefore they are expanded immediatelly before the dependency tree is known internally. Therefore, during this expansion `%` nor `$@` is **not** set and can't be used in the `shell` directive. The workaround is to use the secondary expansion feature (hack), which runs between the 2 phases, when the recipe name is already expanded and the value of `$@` is set. To use this one needs to use the `.SECONDEXPANSION:` rule and rules defined below this rule will be expanded twice. Variables with a single dollar sign will be expanded during the first expansion. When the first expansion sees two dollar signs it interprets it as escaped dollar sign and so it produces a single dollar sign. Then when the second expansion runs variables which are already expanded are treated as text and are left alone, and when it sees single dollar sign (created from the double dollar sign by the previous run) it expands this variable. This way we can force make to have variable `$@` set when calling shell and therefore generate the proper dependency name from recipe name.

Now that we know what we need, let's build it. Building the test file, unity framework and test runner is trivial:

```
$(TEST_OBJS)unity.o:: $(UNITY)unity.c $(UNITY)unity.h
    @echo 'Compiling Unity'
    @$(TEST_CC) $(TEST_CFLAGS) -c $< -o $@

$(TEST_OBJS)%.o:: $(TEST_PATH)%.c
    @echo 'Compiling test $<'
    @mkdir -p `dirname $@`
    @$(TEST_CC) $(TEST_CFLAGS) -c $< -o $@

$(TEST_OBJS)%-runner.o: $(TEST_RUNNERS)%-runner.c
    @echo 'Compiling test runner $<'
    @mkdir -p `dirname $@`
    @$(TEST_CC) $(TEST_CFLAGS) -c $< -o $@
```

Test runner is generated automatically, so in order to build it we must generate it from the test file. Unity provides us with a script for this purpose.

```
$(TEST_RUNNERS)%-runner.c:: $(TEST_PATH)%.c
    @echo 'Generating runner for $@'
    @mkdir -p `dirname $@`
    @ruby $(UNITY)../auto/generate_test_runner.rb $< $@
```

Last thing to do is to build the file under test itself:

```
$(TEST_OBJS)%-under_test.o: $(TEST_ROOT)%.c
    @echo 'Compiling $<'
    @mkdir -p `dirname $@`
    @$(TEST_CC) $(TEST_CFLAGS) -c $< -o $@
    @objcopy --weaken $@
```

The difference is that there is an additional step `objcopy --weaken` for reasons described above.

And that's it, except minor details like creating folders and cleaning up. Several things are a low priority and are not finished:

- There is no `testable_static` macro. It will be added when it is needed.
- There is no dependency management. There is a rule which cleans everything after the tests have run. Proper dependency management is hard as hell, and it is easier and more reliable to rebuild it every time. Tests are built

really fast, executed really fast and can even be executed individually if needed. If this becomes a real problem dependency management can be added.

Note: This is the ugliest piece of Makefile I've ever written and I'm seriously considering rewriting it to different build system. But for now it works, is maintainable and well documented, and there are more pressing things to do (like writing some **actual code for the project**).

Integration testing

I will get around to it once I have some actual modules to test.

RFID Stack

Main functionality of the Reader is to, of course, read an RFID card. To accomplish this it uses a modular stack of libraries written for this purpose. Goals of these libs are:

- To be modular, with clearly defined interfaces
- To fully comply with international standards
- To be well-documented and usable even outside of Deadlock
  - Specifically, its structure should resemble other ChibiOS HAL components and should be easily incorporable to ChibiOS Community HAL
- To be thread safe
- To be easy to use, but not while compromising modularity

Documentation of components

MFRC522 Driver

Driver for the MFRC522 module.

This is a driver for the MFRC522 Mifare and NTAG frontend. It supports the MFRC522 chip connected over various interfaces. It exports an *Pcd* object for use by other layers.

Note: This driver requires the EXT driver to be enabled and configured with non-`const` EXTConfig strcture (as opposed to one documented by function extStart)!

This driver invokes the extSetChannelMode function, see its docs for explanation.

Defines

`MFRC522_USE_SPI`
Enables SPI support.

Note Disabling this option saves both code and data space.

`MFRC522_USE_I2C`
Enables I2C support.

Note Disabling this option saves both code and data space.

MFRC522_USE_UART
Enables UART support.

Note Disabling this option saves both code and data space.

MFRC522_MAX_DEVICES
Maximum number of simultaneously active devices this driver should handle.

Note Lowering this value saves data space and increases driver performance.

Functions

void **mfrc522Init**(void)
Initializes the MFRC522 driver.

Note This function is called implicitly by halCustomInit, no need to call it explicitly.

void **mfrc522ObjectInitSPI**(*Mfrc522Driver* *mdp, SPIDriver *spip)
Initializes the driver object for a MFRC522 module connected over the SPI.

Parameters

- `mdp`: Uninitialized driver object.
- `spip`: Initialized and started SPI driver object.

void **mfrc522ObjectInitI2C**(*Mfrc522Driver* *mdp, I2CDriver *i2cp)
Initializes the driver object for a MFRC522 module connected over the I2C.

Note Not yet implemented!

Parameters

- `mdp`: Uninitialized driver object.
- `i2cp`: Initialized and started I2C driver object

void **mfrc522ObjectInitSerial**(*Mfrc522Driver* *mdp, SerialDriver *sdp)
Initializes the driver object for a MFRC522 module connected over the Serial.

Note Not yet implemented!

Parameters

- `mdp`: Uninitialized driver object.
- `sdp`: Initialized and started Serial driver object

void **mfrc522Start**(*Mfrc522Driver* *mdp, **const** *Mfrc522Config* *config)
Starts the MFRC522 module.

This function powers up, soft-resets the MFRC522 module, initializes, configures it and register is for use with this driver. It also reconfigures the provided Ext driver and register its own interrupt handler to handle interrupts on channel config->interrupt_channel.

Parameters

- `mdp`: Initialized driver object

- `config`: Configuration to apply

void **`mfrc522Reconfig`** (*Mfrc522Driver* *mdp, **const** *Mfrc522Config* *config)

Reconfigures the MFRC522 module without resetting it.

This function reprograms control registers of the MFRC522 module without resetting it, which is useful for hot-swapping MFRC522-specific config options during the module run-time.

This function won't reconfig the following options:

- `EXTDriver *extp`
- `expchannel_t interrupt_channel`
- `void *reset_line`

The only way to change these is to stop and restart the module.

Parameters

- `mdp`: Started driver object
- `config`: Configuration to apply

void **`mfrc522Stop`** (*Mfrc522Driver* *mdp)

Stops the MFRC522 module.

Unregisters this module from this driver and powers it down.

Parameters

- `mdp`: Started driver object

struct `Mfrc522Config`

#include <hal_mfrc522.h> Config options for the MFRC522 module.

In order for the driver to function properly you have to specify:

- `EXTDriver *extp`
- `expchannel_t interrupt_channel`
- `void *reset_line`

Rest of these options are intended for advanced configuration of the module if you have special needs, otherwise default values are just fine.

If you need to modify something consult the MFRC522 Datasheet.

Public Types

enum `driver_input_select_t`

Selects the input of drivers TX1 and TX2.

Default: MFRC522_DRSEL_MPE

MFRC522 Datasheet page 51

Values:

`MFRC522_DRSEL_3STATE` = 0b00

3-state mode during soft powerdown

MFRC522_DRSEL_MPE = 0b01
modulation signal (envelope) from the internal encoder, Miller pulse encoded

MFRC522_DRSEL_MFIN = 0b10
modulation signal (envelope) from pin MFIN

MFRC522_DRSEL_HIGH = 0b11
HIGH; the HIGH level depends on the setting of bits InvTx1RFOn/InvTx1RFOff and InvTx2RFOn/InvTx2RFOff

enum mfout_select_t
Selects the input for pin MFOUT.

Default: MFRC522_MFSEL_3STATE

MFRC522 Datasheet page 52

Values:

MFRC522_MFSEL_3STATE = 0b0000
3-state

MFRC522_MFSEL_LOW = 0b0001
LOW.

MFRC522_MFSEL_HIGH = 0b0010
HIGH.

MFRC522_MFSEL_TBUS = 0b0011
test bus signal as defined by the test_bus_bit_sel

MFRC522_MFSEL_MPE = 0b0100
modulation signal (envelope) from the internal encoder, Miller pulse encoded

MFRC522_MFSEL_SSTRT = 0b0101
serial data stream to be transmitted, data stream before Miller encoder

MFRC522_MFSEL_SSTRR = 0b0111
serial data stream received, data stream after Manchester decoder

enum cl_uart_in_sel_t
Selects the input of the contactless UART.

Default: MFRC522_UINSEL_ANALOG

MFRC522 Datasheet page 52

Values:

MFRC522_UINSEL_LOW = 0b00
constant LOW

MFRC522_UINSEL_MAN_MFIN = 0b01
Manchester with subcarrier from pin MFIN

MFRC522_UINSEL_ANALOG = 0b10
modulated signal from the internal analog module, default

MFRC522_UINSEL_NRZ_MFIN = 0b11
NRZ coding without subcarrier from pin MFIN which is only valid for transfer speeds above 106 kBd

enum receiver_gain_t
Gain of the receiver.

Default: MFRC522_GAIN_33

MFRC522 Datasheet page 59

Values:

MFRC522_GAIN_18 = 0b000
18 dB

MFRC522_GAIN_23 = 0b001
23 dB

MFRC522_GAIN_33 = 0b100
33 dB

MFRC522_GAIN_38 = 0b101
38 dB

MFRC522_GAIN_43 = 0b110
43 dB

MFRC522_GAIN_48 = 0b111
48 dB

Public Members

EXTDriver ***extp**
EXT driver to use for handling MFRC522-issued interrupts

expchannel_t **interrupt_channel**
EXT channel to which the IRQ pin (**and only the IRQ pin**) of the MFRC522 is connected.

void ***reset_line**
PAL line, which when set low resets the connected MFRC522

bool **MFIN_polarity**
Defines polarity of pin MFIN.

true: MFIN is active HIGH false: MFIN is active LOW

Default: true

MFRC522 Datasheet page 48

bool **inverse_modulation**
Modulation of transmitted data should be inverted.

Default: false

MFRC522 Datasheet page 49

uint8_t **tx_control_reg**
Value of the transmission control register.

Default: 0x80

MFRC522 Datasheet page 50

Mfrc522Config::driver_input_select_t **driver_input_select**

Mfrc522Config::mfout_select_t **mfout_select**

Mfrc522Config::cl_uart_in_sel_t **cl_uart_in_sel**

uint8_t **min_rx_signal_strength**
Minimum signal strength which will be accepted by the decoder.

Only 4 lowest bits are taken into account.

Default: 8

MFRC522 Datasheet page 53

uint8_t **min_rx_collision_level**
Minimum collision signal strength.

Minimum signal strength at the decoder input that must be reached by the weaker half-bit of the Manchester encoded signal to generate a bit-collision relative to the amplitude of the stronger half-bit.

Only 3 lowest bits are taken into account.

Default: 4

MFRC522 Datasheet page 53

uint8_t **demod_reg**
Demodulator settings.

Default: 0x4D

MFRC522 Datasheet page 53

Mfrc522Config::receiver_gain_t **receiver_gain**

uint8_t **transmit_power_n**
Conductance of the output n-driver (CWGsN) which can be used to regulate the output power.

Default: 8

MFRC522 Datasheet page 59

uint8_t **modulation_index_n**
Conductance of the output n-driver (ModGsN) which can be used to regulate the modulation index.

Default: 8

MFRC522 Datasheet page 59

uint8_t **transmit_power_p**
Conductance of the output p-driver (CWGsP) which can be used to regulate the output power.

Default: 32

MFRC522 Datasheet page 60

uint8_t **modulation_index_p**
Conductance of the output n-driver (ModGsP) which can be used to regulate the modulation index.

Default: 32

MFRC522 Datasheet page 60

struct Mfrc522Driver
#include <hal_mfrc522.h> Structure representing a MFRC522 driver.

For functions expecting a *Pcd* object use the `pcd` memeber of this structure. Otherwise don't modify any of these values.

Public Types

enum **mfrc522_conntype_t**
How is the MFRC522 connected.

Values:

MFRC522_CONN_SPI

MFRC522_CONN_I2C

MFRC522_CONN_SERIAL

Public Members

Pcd **pcd**
Abstract *Pcd* structure to be used with other parts of this stack

pcdstate_t **state**
Driver state

Mfrc522Driver::mfrc522_conntype_t **connection_type**

union *Mfrc522Driver*::*iface_u* **iface**

EXTDriver ***extp**
EXT driver to use for handling MFRC522-issued interrupts

expchannel_t **interrupt_channel**
EXT channel to which the IRQ pin is connected

void ***reset_line**
PAL line, which when set low resets the connected MFRC522

const *Mfrc522Config* ***current_config**
Lastly applied config

volatile bool **interrupt_pending**
Interrupt is pending for this reader

thread_reference_t **tr**
Thread reference the reader will sleep on

uint8_t **response**[64]
Response buffer

uint8_t **resp_last_valid_bits**
Number of last valid bits in the response

uint8_t **resp_length**
Response length

uint8_t **resp_read_bytes**
Number of already retreived response bytes

mutex_t **mutex**
Mutex for mutual access

union **iface_u**
#include <*hal_mfrc522.h*> Pointer to a given interface driver.

Public Members

SPIDriver ***spip**

I2CDriver ***i2cp**

SerialDriver ***sdp**

Internal functioning of the driver

Although most of the drivers in the HAL are contained in one file, the MFRC522 driver was becoming unreadable, so I've split it between several files.

- `src/hal_mfrc522/hal_mfrc522_internal.h` header contains internal constants and internal documentation. It is not supposed to be used by application using this driver.
- `src/hal_mfrc522/hal_mfrc522.c` is the main driver file. It contains MFRC522-specific initialization and configuration functions. It also contains the interrupt handler only purpose of which is to wake up a sleeping thread.
- `src/hal_mfrc522/hal_mfrc522_ext_api.c` contains implementation of extended features. See hal_abstract_iso14443_pcd_ext.h.
- `src/hal_mfrc522/hal_mfrc522_llcom.c` contains low-level communication routines for reading and writing registers of the MFRC522 over various connection interfaces.
- `src/hal_mfrc552/hal_mfrc522_pcd_api.c` contains implementation of *Pcd* API functions for MFRC522.

Primary goals

This driver should provide easy-to-use synchronous API to drivers of higher protocol layers while being efficient and friendly to other threads (it suspends the calling thread while it's waiting for data).

Initialization and configuration functions handling global objects of this driver must be thread-safe.

This driver does not guarantee thread safety if single *Mfrc522Driver* file is used simultaneously by multiple threads. Those threads should call *pcdAcquireBus* and *pcdReleaseBus* to achiveve mutual exclusion. However, it is safe to use different *Mfrc522Driver* objects from different threads simultaneously.

This driver should be as universal as possible. This is the reason for (over?)complicated configuration structure. It allows you to, when you know what are you doing, configure the MFRC522 modulee for your specific use-case (like using an external modulator with the chip). It is also easy to use, since default values work out-of-the box with the typical use-case (like the RFID-RC522 chinese module).

Thread suspend and interrupt handling

Each time this driver waits for the action of the reader it suspends the calling thread, to allow other threads to run.

MFRC522 has a mechanism of waking up the host using its IRQ pin. Host can conigure which interrupts propagate to the interrupt pin, and later figure out which interrupt occured by reading a specific register.

This driver uses the Ext driver provided by ChibiOS to handle these interrupts. It registers its own interrupt handler, the same for each interrupt channel. However, in the current version the Ext driver doesn't allow passing custom parameters to the handler function, therefore when an interrupt occurs we only know which channel it came from. Then we may wake up all sleeping driver threads and let every thread check whether the interrupt is intended for it. This is problemating due to race conditions, necessity of events buffering, etc. Instead, we have imposed limitation that each interrupt channel can have only one reader (and nothing else than the reader) attached to it. Therefore when an interrupt occures by knowing the interrupt channel we can wake up the proper thread.

In addition this wake-up is buffered if the thread couldn't be suspended in time.

Anticollision detection

TODO

ISO/IEC 14443 PCD Abstract Class

Abstract ISO14443 Proximity Coupling Device (card reader) driver interface.

This header defines an abstract interface useful to access generic ISO14443-compliant PCDs (contactless card readers) in a standardized way.

Design

This header provides communication interface with the PCD (Proximity Coupling Device) as described in `ISO/IEC 14443-3`. This standard defines communication with 2 different card types: `A` and `B`. The naming convention is 'A'-related functions end with A, B-related functions end with B, common functions end with AB.

Currently, API for part A is fully designed, API for part B will be added later.

Communication between the PCD and the PICC consists of sending and receiving frames. The frames are transmitted in pairs, PCD to PICC followed by PICC to PCD.

Part A of the standard defines transmission of 3 different types of frames:

- Short frame: transmits 7 bits.
- Standard frame: Used for data exchange and can transmit several bytes with parity.
- Bit oriented anticollision frame: 7 byte long frame spit anywhere into two parts. First part is transmitted by the PCD, second part is added by the PICC. It is used during bit-oriented anticollision loop.

ISO/IEC 14443-3 specifies different communication methods (different modulation type / index, different encoding) for part A and B. This driver should support setting these modes, various other communication parameters within these modes as defined by the standard, and should also be able to advertise its capabilities. Communication speeds can't be arbitrary and are defined by ISO/IEC 14443-4 as `$ 1 etu = 128 / (D x fc) $`, where etu is the elementary time unit (duration of one bit), fc is the carrier frequency (defined in ISO/IEC 14443-2 to be `$ 13.56MHz \pm 7kHz $`) and `D` is an integer divisor, which may be 1, 2, 4 or 8. This paradoxically means that increasing the divisor also increases the communication speed.

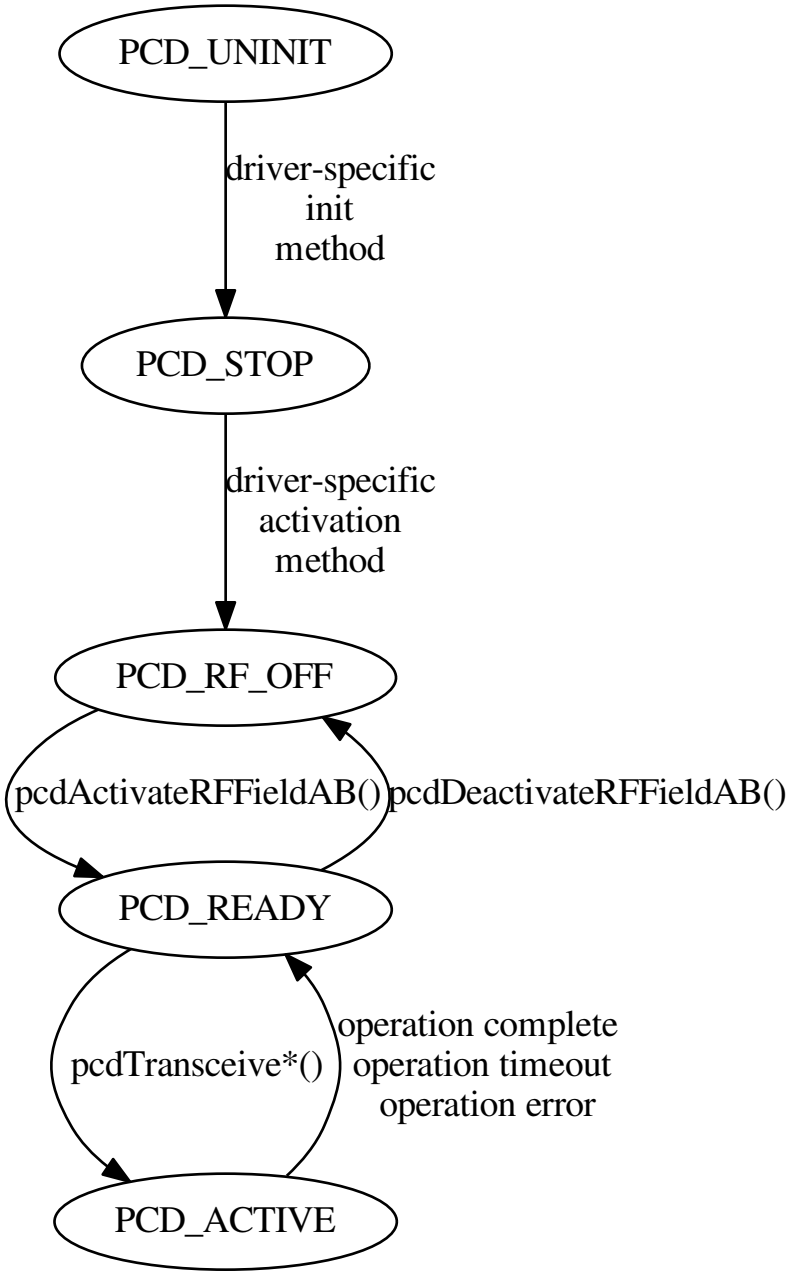
The readers also usually support a number of extended features, not covered by the ISO/IEC 14443 standard. For example, the MFRC522 is able to perform a Mifare authentication using its crypto unit, or a self-test. Upper layers which know how to use these extended features should have access to them, but they should not clutter the main API. That is why each extended feature will have a globally assigned number (in the global abstract header). Then other layers could use these numbers to invoke the extended feature, passing in a "parameter structure". These structures are also defined globally in the abstract header (although in a different file) to allow their re-use.

Often PCDs have a maximum data size they can handle at once. ISO/IEC 14443 standard takes this into account and defines "protocol chaining", a method to send large data units in multiple smaller frames. The upper library handles this, but for it to know whether to use chaining the PCD must be able to report maximum frame size it can handle.

Driver state diagram

This abstract class presumes a driver with state. The following diagram illustrates available states (see *pcdstate_t*) and transitions between them.

Some functions may be called only in specific states, this is indicated in documentation of each function. Calling a function in an invalid state will be caught by assertion (if assertions are enabled).



Thread safety

Implementation of this API does not have to guarantee that it is thread-safe. If you need API access from multiple threads use *pcdAcquireBus* and *pcdReleaseBus* APIs in order to get exclusive access.

Macro functions (Pcd)

Convenience macros for easy calling of 'member functions'

pcdGetStateAB(ip) **See** *BasePcdVMT.getStateAB*

pcdActivateRFAB(ip) **See** *BasePcdVMT.activateRFAB*

pcdDeactivateRFAB(ip) **See** *BasePcdVMT.deactivateRFAB*

pcdGetSupportedParamsAB(ip) **See** *BasePcdVMT.getSupportedParamsAB*

pcdSetParamsAB(ip, rx_spd, tx_spd, mode, txcrc, rxcrc) **See** *BasePcdVMT.setParamsAB*

pcdTransceiveShortFrameA(ip, data, resp_len_p, timeout_us) **See** *BasePcdVMT.transceiveShortFrameA*

pcdTransceiveStandardFrameA(ip, data, size, resp_len_p, timeout_us) **See** *BasePcdVMT.transceiveStandardFrameA*

pcdTransceiveAnticollFrameA(ip, data, size, n_last_bits, align_rx, resp_len_p, timeout_us) **See** *BasePcdVMT.transceiveAnticollFrameA*

pcdGetRespLengthA(ip) **See** *BasePcdVMT.getResponseLengthA*

pcdGetRespAB(ip, buf_size, buffer, size_copied, n_last_bits_p) **See** *BasePcdVMT.getResponseAB*

pcdAcquireBus(ip) **See** *BasePcdVMT.acquireBus*

pcdReleaseBus(ip) **See** *BasePcdVMT.releaseBus*

pcdSupportsExtFeature(ip, feature) **See** *BasePcdVMT.supportsExtFeature*

pcdCallExtFeature(ip, feature, params, result) **See** *BasePcdVMT.callExtFeature*

Enums

enum pcdstate_t

States of the driver state machine.

Values:

PCD_UNINT
Not initialized

PCD_STOP
Initialized, not active.

PCD_RF_OFF
RF Field is off

PCD_READY
Ready to transmit

PCD_ACTIVE
Transceiving

enum pcdresult_t
Operation result codes.

Values:

PCD_OK
Command completed successfully

PCD_BAD_STATE
Command not possible in this state

PCD_UNSUPPORTED
This PCD does not support this command

PCD_OK_COLLISION
Command OK, but received collision

PCD_OK_TIMEOUT
Command OK, but card did not respond

PCD_ERROR
An unspecified error has occured

PCD_TX_ERROR
Transmission error

PCD_RX_ERROR
Receiver error (such as bad parity)

PCD_RX_OVERFLOW
A receive buffer has overflown

PCD_TX_OVERFLOW
This message won't fit to tx buffer

enum pcdspeed_rx_t
Receive speed keys for speed bitmask.

Values:

PCD_RX_SPEED_106 = 1

PCD_RX_SPEED_212 = 2

PCD_RX_SPEED_424 = 4

PCD_RX_SPEED_848 = 8

enum pcdspeed_tx_t
Transmit speed keys for speed bitmask.

Values:

PCD_TX_SPEED_106 = 16

PCD_TX_SPEED_212 = 32

PCD_TX_SPEED_424 = 64

PCD_TX_SPEED_848 = 128

enum pcdmode_t
Standard communication modes.

Values:

PCD_ISO14443_A = 1

PCD_ISO14443_B = 2

enum pcdfeature_t
List of possible extended features.

For each extended feature 2 structures are defined in file hal_abstract_iso14443_pcd_ext.h. The first is a param strucure, the second is a response structure.

Values:

PCD_EXT_SELFTEST
Perform a self-test

PCD_EXT_CALCULATE_CRC_A
Calculate type-A CRC

PCD_EXT_CALCULATE_CRC_B
Calculate type-B CRC

PCD_EXT_MIFARE_AUTH
Perform a Mifare auth and turn on crypto

struct PcdSParams
#include <hal_abstract_iso14443_pcd.h> Structure of communuication parameters supported by the PCD.

Public Members

uint8_t **supported_speedsA**
Bit mask of supported tx/rx speeds

uint8_t **supported_speedsB**
Bit mask of supported tx/rx speeds

bool **supported_asym_speeds**
Support of asymetric speed setting

uint8_t **supported_modes**
Bit mask of supported modes (A or B)

uint8_t **supported_crc_on**
Support for automatic CRC generation

uint8_t **supported_crc_off**
Whether CRC gen can be off at the given speed

uint16_t **max_tx_size**
Maximum Transmit buffer size

uint16_t **max_rx_size**
Maximum Receive buffer size

struct BasePcdVMT
#include <hal_abstract_iso14443_pcd.h> Abstract ISO14443 PCD Virtual Method Table.

Note use macros defined in this file to call these functions for convenience. Name of the macro is the same as a name of this function with `pcd` prefix.

Public Members

pcdstate_t (***getStateAB**) (void *inst)
Returns the device state.

Note This function can be called in all states.

Return State of the device

Parameters

- `inst`: Pointer to a *Pcd* structure

pcdresult_t (***activateRFAB**) (void *inst)
Activates RF Field of the PCD.

Note This function can be called in the following states:

- *PCD_RF_OFF*

Parameters

- `inst`: Pointer to a *Pcd* structure

Return Value

- `PCD_OK`: RF Field successfully activated. *Pcd* transitions to *PCD_READY* state.
- `PCD_BAD_STATE`: RF Field cannot be activated now or is already active. State won't change.
- `PCD_ERROR`: An error has occured. State won't change.

pcdresult_t (***deactivateRFAB**) (void *inst)
Deactivates the RF Field of the PCD.

Note This function can be called in the following states:

- *PCD_READY*

Parameters

- `inst`: Pointer to a *Pcd* structure

Return Value

- `PCD_OK`: Field successfully deactivated. *Pcd* transitions to *PCD_RF_OFF* state.
- `PCD_BAD_STATE`: RF Field cannot be deactivated now or is already inactive. State won't change.
- `PCD_ERROR`: An error has occured. State won't change.

const *PcdSParams* *(***getSupportedParamsAB**) (void *inst)
Returns structure with supported features of this PCD.

Note This function can be called in all states

Return Pointer to a *PcdSParams* structure. Content of this structure should not be modified.

Parameters

- `inst`: Pointer to a *Pcd* structure

pcdresult_t (***setParamsAB**)(void *inst, *pcdspeed_rx_t* rx_spd, *pcdspeed_tx_t* tx_spd, *pcdmode_t* mode, bool generate_CRC, bool verify_CRC)

Sets communication parameters.

If the parameter combination is unsupported *PCD_UNSUPPORTED* will be returned. It is advised to first check the value returned by *getSupportedParamsAB*.

Note This function can be called in the following states:

- *PCD_READY*
- *PCD_RF_OFF*

Parameters

- `inst`: pointer to a *Pcd* strucutre
- `rx_spd`: desired reception speed
- `tx_spd`: desired transmission speed
- `mode`: desired communication mode
- `generate_CRC`: Whether the CRC should be generated and transmitted automatically during transmission
- `verify_CRC`: Whether the CRC should be verified automatically during reception

Return Value

- `PCD_OK`: Parameters applied successfully. State won't change.
- `PCD_BAD_STATE`: Parameters can't be changed now. State won't change.
- `PCD_UNSUPPORTED`: Some requested parameters are not supported byt this PCD. State won't change.
- `PCD_ERROR`: An error has occured. State won't change.

pcdresult_t (***transceiveShortFrameA**)(void *inst, uint8_t data, uint16_t *resp_length, uint32_t timeout_us)

Transmits a 'Short Frame' and blocks until the response is received or a timeout occurs.

Short frame transmits 7 data bits without parity. Therefore only 7 Least Significant Bits of parameter `data` are sent. This function discards remaining data in the response buffer, if any.

Note This function can be called in the following states:

- *PCD_READY*

Note This function call either returns immediatelly with error or will change the state *PCD_ACTIVE* when the operation is in progress. Unless noted otherwise, the state returns to *PCD_READY* after this function returns.

Parameters

- `inst`: pointer to a *Pcd* structure
- `data`: 7 bits of data to be sent (MSB is ignored)
- `resp_len_p`: Length of the received response (see *transceiveStandardFrameA*).

- `timeout_us`: Max number of microseconds to wait for a response.

Return Value

- `PCD_OK`: Transmission successful and response received.
- `PCD_OK_COLLISION`: Transmission successful and multiple responses received.
- `PCD_OK_TIMEOUT`: Transmission successful but no response was received.
- `PCD_BAD_STATE`: Can't transmit right now. The state won't change.
- `PCD_ERROR`: A driver od hardware error has occured.
- `PCD_TX_ERROR`: A transmission error has occured.
- `PCD_RX_ERROR`: A reception error has occured.
- `PCD_RX_OVERFLOW`: Too much data received.

pcdresult_t (***transceiveStandardFrameA**)(void *inst, uint8_t *buffer, uint16_t length, uint16_t *resp_length, uint32_t timeout_us)

Transmits a 'Standard Frame' and blocks until the response is ready.

Standard frame transmits `n` (where `n >= 1`) bytes. The `len` parameter cannot be greater than maximum buffer size supported by the reader. After the response is received `resp_len_p` will be set to the received response length. You can then use this parameter to allocate a new buffer and obtain the response using the *getResponseAB* function. This function discards remaining data in the response buffer, if any.

Note This function can be called in the following states:

- *PCD_READY*

Note This function call either returns immediatelly with error or will change the state *PCD_ACTIVE* when the operation is in progress. Unless noted otherwise, the state returns to *PCD_READY* after this function returns.

Parameters

- `inst`: pointer to a *Pcd* structure
- `buffer`: Bytes to send in a standard frame
- `len`: Size of the `buffer`
- `resp_len_p`: Size of received response
- `timeout_us`: Max number of microseconds to wait for a response.

Return Value

- `PCD_OK`: Transmission successful and response received.
- `PCD_OK_COLLISION`: Transmission successful and multiple responses received.
- `PCD_OK_TIMEOUT`: Transmission successful but no response was received.
- `PCD_BAD_STATE`: Can't transmit right now. The state won't change.
- `PCD_ERROR`: A driver od hardware error has occured.
- `PCD_TX_ERROR`: A transmission error has occured.
- `PCD_RX_ERROR`: A reception error has occured.
- `PCD_RX_OVERFLOW`: Too much data received.
- `PCD_TX_OVERFLOW`: Can't transmit this much data. The state won't change.

pcdresult_t (***transceiveAnticollFrameA**) (void *inst, uint8_t *buffer, uint16_t length, uint8_t n_last_bits, uint8_t align_rx, uint16_t *resp_length, uint32_t timeout_us)

Transmits the first part of an 'Anticollision Frame' and blocks until the response is ready.

Anticollision frame is a standard 7-byte frame split anywhere after 16th bit and before 55th bit. First part is transmitted by the PCD and the second part is transmitted by the PICC as a part of an anticollision sequence. For `resp_len_p` meaning see pcdTransceiveStdFrameA. This function discards remaining data in the response buffer, if any.

Note This function can be called in the following states:

- *PCD_READY*

Note This function call either returns immediatelly with error or will change the state *PCD_ACTIVE* when the operation is in progress. Unless noted otherwise, the state returns to *PCD_READY* after this function returns.

Parameters

- `inst`: pointer to a *Pcd* structure
- `buffer`: Bytes to send in an anticoll frame
- `len`: Size of the buffer
- `n_last_bits`: Number of valid bits in the last byte to be transmitted. 0 means the whole byte is valid.
- `align_rx`: Desired position of the first received bit in the first byte of the response.
- `resp_len_p`: Size of the received response
- `timeout_us`: Max number of microseconds to wait for a response.

Return Value

- `PCD_OK`: Transmission successful and response received.
- `PCD_OK_COLLISION`: Transmission successful and multiple responses received.
- `PCD_OK_TIMEOUT`: Transmission successful but no response was received.
- `PCD_BAD_STATE`: Can't transmit right now. The state won't change.
- `PCD_ERROR`: A driver od hardware error has occured.
- `PCD_TX_ERROR`: A transmission error has occured.
- `PCD_RX_ERROR`: A reception error has occured.
- `PCD_RX_OVERFLOW`: Too much data received.
- `PCD_TX_OVERFLOW`: Can't transmit this much data. The state won't change.

uint16_t (***getResponseLengthA**) (void *inst)

Gets (remaining) size of response stored in the buffer, if any.

Note This function can be called in the following states:

- *PCD_READY*
- *PCD_RF_OFF*

Return Number of bytes in the response buffer. `0` if the response buffer is empty.

Parameters

- `inst`: pointer to a *Pcd* structure

pcdresult_t (***getResponseAB**) (void *inst, uint16_t buffer_size, uint8_t *buffer, uint16_t *size_copied, uint8_t *n_last_bits)

Read response from the internal response buffer.

Read response from the internal buffer. If size of the buffer passed to this function is smaller than the response size only part of the response will be copied and this function must be called several times. Bytes received first are copied first. `n_last_bits_p` is valid only when copying last part of the response.

Note This function can be called in the following states:

- *PCD_READY*
- *PCD_RF_OFF*

Parameters

- `inst`: pointer to a *Pcd* structure
- `buf_size`: Max size to read
- `buffer`: The buffer
- `size_copied`: Number of bytes written to the buffer
- `n_last_bits_p`: Number of valid bits in last byte copied to the buffer.

Return Value

- `PCD_OK`: Bytes were copied
- `PCD_ERROR`: No more bytes to copy
- `PCD_BAD_STATE`: This function can't be called in this state.

void (***acquireBus**) (void *inst)

Acquires exclusive access to the PCD.

Parameters

- `inst`: pointer to a *Pcd* structure

void (***releaseBus**) (void *inst)

Acquires exclusive access to the PCD.

Parameters

- `inst`: pointer to a *Pcd* structure

bool (***supportsExtFeature**) (void *inst, *pcdfeature_t* feature)

Checks whether this PCD supports the given extended feature.

Parameters

- `inst`: pointer to a *Pcd* structure

pcdresult_t (***callExtFeature**) (void *inst, *pcdfeature_t* feature, void *params, void *result)

Invokes an extended feature.

Parameters

- `inst`: pointer to a *Pcd* structure

- `feature`: identifier of the feature
- `params`: parameters passed to the feature function. Can be NULL if function doesn't expect any.
- `result`: result of the feature function.

Return Value

- `PCD_OK`: Feature function executed successfully
- `PCD_BAD_STATE`: Feature function can't be executed now
- `PCD_UNSUPPORTED`: This feature is not supported

struct `Pcd`

#include <hal_abstract_iso14443_pcd.h> Base ISO/IEC 14443 PCD.

This class represents a generic ISO/IEC 14443 Proximity Coupling device.

Public Members

const struct *BasePcdVMT* ***vmt**

Virtual Methods Table

void ***data**

Private data of a driver

ISO/IEC 14443 PCD Structures for Extended Features

Structures for extended / optional features of an abstract ISO/IEC 14443 PCD.

Real-world ISO/IEC 14443 PCDs (card readers) support a number of extended features which are not covered by the abstract PCD driver. These features can be still used using the pcdCallExtraFeatures API. Structures used as parameters and results of various commands are defined here.

struct `PcdExtSelftest_params`

#include <hal_abstract_iso14443_pcd_ext.h> Parameters for *PCD_EXT_SELFTEST* command.

PCD_EXT_SELFTEST takes no parameters, this structure is empty and `params` argument can be `NULL`.

struct `PcdExtSelftest_result`

#include <hal_abstract_iso14443_pcd_ext.h> Result of the *PCD_EXT_SELFTEST* command.

Public Members

bool **passed**

Did the self-test passed?

struct `PcdExtCalcCRC_params`

#include <hal_abstract_iso14443_pcd_ext.h> Parameters for *PCD_EXT_CALCULATE_CRC_A* and *PCD_EXT_CALCULATE_CRC_B* commands.

Public Members

uint16_t **length**
Size of the data to calculate CRC of

uint8_t *__buffer__
Pointer to data buffer

struct PcdExtCalcCRC_result
#include <*hal_abstract_iso14443_pcd_ext.h*> Result of *PCD_EXT_CALCULATE_CRC_A* and *PCD_EXT_CALCULATE_CRC_B* commands.

Public Members

uint16_t **crc**
Resulting CRC

struct PcdExtMifareAuth_params
#include <*hal_abstract_iso14443_pcd_ext.h*> Parameters for the *PCD_EXT_MIFARE_AUTH* command.

Public Members

uint8_t **authCommandCode**

uint8_t **blockAddr**

uint8_t **sectorKey**[6]

uint8_t **cardSerialNumber**[4]

struct PcdExtMifareAuth_result
#include <*hal_abstract_iso14443_pcd_ext.h*> Result of the *PCD_EXT_MIFARE_AUTH* command.

Public Members

bool **authSuccess**

Abstract Command-Response Card Object

Abstract Command-Response Card object.

This header defines an abstract interface used to communicate with an Integrated Circtuit Card (either with contacts or contactless) using request-response frames. Each frame is a request and should generate some response frame or a timeout. Example of such a card is ISO/IEC 14443 Proximity Integrated Circuit Card or ISO/IEC 7816 Integrated Circuit Card exchanging command-response pairs using T=0 or T=1 protocol.

For now only synchronous API is defined. Async API may be added later if needed.

Macro functions (Pcd)

Convenience macros for easy calling of 'member functions'

crcardTransceive(inst, tx_buf_p, tx_buf_size, resp_size_p) **See**
CRCardVMT.transceive

crcardGetResponseSize (inst) **See** *CRCardVMT.getResponseSize*

crcardGetResponse (inst, buf_p, max_buf_size) **See** *CRCardVMT.getResponse*

Enums

enum crcard_result_t
Operation result codes.

Values:

CRCARD_OK
Transmission successful, received response

CRCARD_TX_ERROR
Unrecoverable transmission errror

CRCARD_RX_ERRPR
Unrecoverable reception error

CRCARD_TIMEOUT
Transmission successful, no response

CRCARD_NONEXISTENT
Card removed, no further comm possible

struct CRCardVMT
#include <hal_abstract_CRCard.h> Abstract *CRCard* Virtual Method Table.

Note use macros defined in this file to call these functions for convenience. Name of the macro is the same as a name of this function with `crcard` prefix.

Public Members

crcard_result_t (***transceive**) (void *inst, uint8_t *tx_buffer, uint16_t tx_buffer_size, uint16_t *response_size)
Send a frame of data to a card and wait for a response.

The response will be stored in a response buffer. Invoking this function will clear contents of the response buffer, if any.

Return Operation result

Parameters

- `inst`: Pointer to a *CRCard* structure
- `tx_buffer`: Data to send
- `tx_buffer_size`: Number of bytes to send
- `response_size`: Number of bytes received

uint16_t (***getResponseSize**) (void *inst)
Get number of remaining bytes in the response buffer.

Return Number of bytes in the response buffer

Parameters

- `inst`: Pointer to a *CRCard* structure

uint16_t (***getResponse**) (void *inst, uint8_t *data, uint16_t buffer_size)
Retrieves a response from the response buffer.

Think of the response buffer as a queue. When this function is called no more than `buffer_size` bytes are removed from this queue and copied to `data` buffer.

Function *getResponseSize* returns number of bytes in this queue.

Return Number of copied bytes

Parameters

- `inst`: Pointer to a *CRCard* structure
- `data`: Buffer to copy the response to
- `buffer_size`: Maximum number of bytes to copy

struct `CRCard`
#include <hal_abstract_CRCard.h> Base Command-Response Card.

This class represents a generic Command-Response Card.

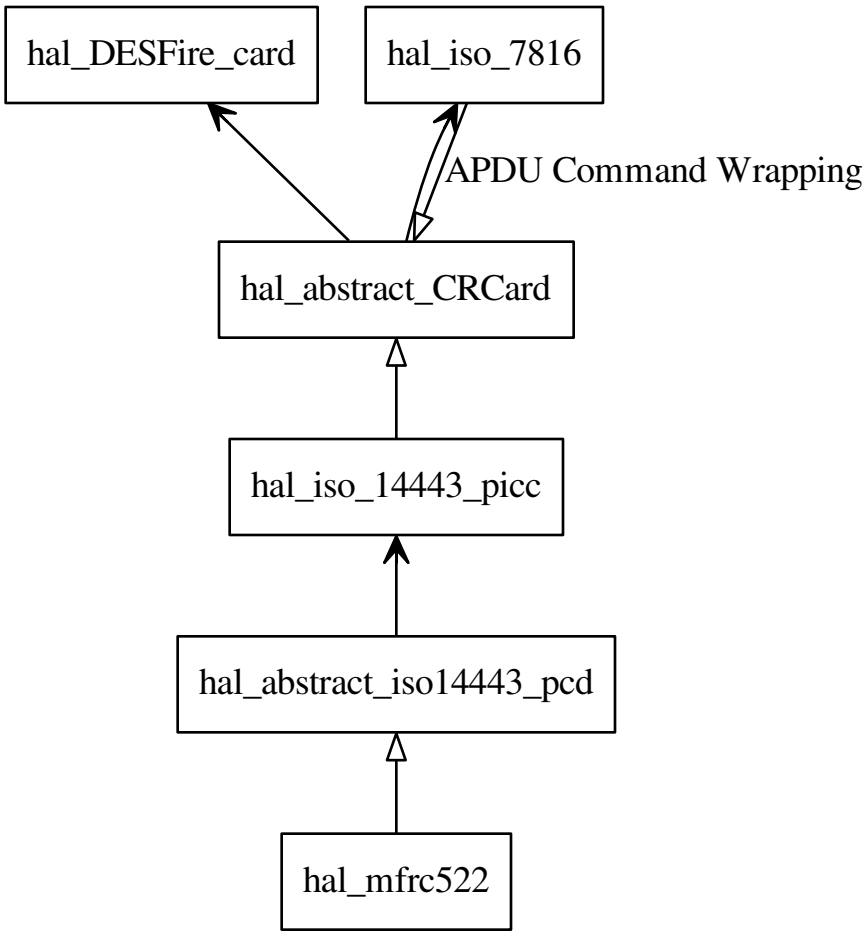
Public Members

const struct *CRCardVMT* ***vmt**
Virtual Methods Table

void ***data**
Private data of a driver

Architecture

Currently the plan is to support primarily ISO/IEC 14443A-compatible cards. The following diagram shows the architecture, currently implemented and planned components:



Components

Abstract classes:

- `hal_abstract_iso14443_pcd`: Abstract representation of a RFID module capable of reading ISO/IEC 14443-compliant cards
- `hal_abstract_CRCard`: Abstract representation of either contactless card or card with contacts which communicates by exchanging Command - Response pairs (where every command generates either an response or a timeout).

Protocol implementations:

- `hal_iso_14443_picc`: Implements initialization, anticollision and communication protocol with ISO/IEC 14443-compliant card.
- `hal_iso_7816`: (Planned) implements industry-standard commands and wrapping of other protocols into its APDUs (and therefore creating another instance of `CRCard` where command-response pairs are wrapped in its APDUs)

- `hal_DESFire_card`: (Planned) implements proprietary DESFire command set.

Device drivers:

- `hal_mfrc522`: Driver for the MFRC522 module.

Abstract classes in Plain C

The RFID stack is written in plain C. That does not mean it can't be written in an object-oriented way.

"Object" in this case is a structure which contains a pointer to another structure, called a Virtual Method Table and arbitrary other private data. The virtual method table is a structure containing pointers to "member functions". These functions, by convention, expect instance of the "object" structure as a first parameter.

To call a "member function" of an "object" you then just would do the following:

```
result = obj->vmt->funct(obj, param1, param2);
```

However, this is cumbersome to write, so libraries which define abstract classes also include a convenience macro for each function:

```
#define funct(obj, param1, param2) ((obj)->vmt->funct(obj, param1, param2))
```

so then you will just write

```
result = funct(obj, param1, param2)
```

To "extend" such an abstract class you just have to define all functions in the Virtual Method Table and provide a function which instantiates this table and instantiates structure which represents the object:

```
typedef struct {
    uint8_t accumulator;
} my_privatedata;

// Definition of a member method
result_t my_funct(FunnyObj obj, uint8_t param1, uint8_t param2) {
    ((my_privatedata*)(obj->data))->accumulator += param1;
    ((my_privatedata*)(obj->data))->accumulator += param2;
    return ((my_privatedata*)(obj->data))->accumulator;
}

// Something like a constructor
FunnyObj my_create_instance(uint8_t init_acc) {
    FunnyObj obj = malloc(sizeof(FunnyObj));
    obj->vmt = malloc(sizeof(FunnyObjVMT));
    obj->data = malloc(sizeof(my_privatedata));

    obj->vmt->funct = &my_funct;
    ((my_privatedata*)(obj->data))->accumulator = 0;
}
```

Of course you could do some more preprocessor magic, but that would violate the principle of least astonishment. This by itself borders on violating it, however, it is the only way to write truly modular protocol stack which allows for easy component swapping and using multiple components of the same type (e.g. reader drivers) simultaneously.

B

C

D

M

P

R