embedded-code-patterns Documentation

kaeraz

Feb 12, 2019

Introduction

This is a blog which deals with code desing issues. It brings ideas how to ogranize the code and how to handle complex tasks in microcontrollers world. This is the iniative of mine dictated by the necessity of structuring and organizing the lessons learned from different projects I took part it. On the other hand this blog delivers articles about various topics such as programming technics, development styles, organization issues, and many other subjects important in coders society.

CHAPTER 1

About the authors

kaeraz or **k2**

> *Programming enthusiast, hacker with great experience that comes from home appliance applications like washing machines, dishwashers and dryers. Focused mostly on multiboard bare-metal embedded systems with 16/32-bit microcontrollers or small RTOS solutions like FreeRTOS...*

1.1 License

1.1.1 GNU General Public License

Version 2, June 1991 Copyright © 1989, 1991 Free Software Foundation, Inc. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: **(1)** copyright the software, and **(2)** offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- **a)** You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- **b)** You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- **c)** If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- **a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- **b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- **c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence

you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands *show w* and *show c* should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than *show w' and 'show c*; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

1.2 Memory allocation using Pool

1.2.1 Introduction

Dynamic memory allocation is a method which reserves RAM memory space (heap) in run-time code execution. In contrast to static memory allocation, it brings new possibilities in case of writing embedded programs, such us:

- in C programming hiding types information (encapsulation) e.g. constructing Abstract Data Types (ADT)
- reserving space for objects while program is running e.g. in protocol stacks
- part of the code may allocate memory, while other part may use it
- making structures that can change its size in time e.g. lists, arrays, maps, queues
- smaller memory size is needed - application uses exactly the space it requires

There are of course many disadvantages of using dynamic memory allocation.

- programmer has to keep track of allocated memory and deallocate it when needed
- it takes time for allocation - this can be indeterministic time
- application can run out of memory
- memory footprint cannot be determined at link time

The following means are used for the purpose of dynamic memory allocation/deallocation:

- in C - functions `malloc` and `free`
- in C++ - operations `new` and `delete` (both with some variations)

Nevertheless, using of the built-in means for small embedded systems is not a good choice. One should think about alternatives due to following reasons:

1. provided toolchain may not provide such possibility (supporting for `malloc` is not available)
2. allocation is not deterministic (time for memory chunk allocation)
3. using built-in `malloc` produces visible extend of the footprint

There are many technics how to overcome issues related with the dynamic memory allocation, but to be honest there is no cure-all solution. You can find some ideas and useful guidelines how to safely allocate memory in embedded world.

1.2.2 Pool pattern

I wrote this article not to elaborate on memory allocation, but to provide you a sample implementation of the quite useful pattern, which is a **Pool** pattern. Let's start...

First of all, a **Pool** is a container that comprises declared `SIZE` number of elements. Each element has the same byte size. The application can take (allocate) elements from the **Pool**, use it and when it is not needed anymore, give it back to the **Pool** (deallocate). Other variation of the **Pool** pattern is when deallocation is not supported. It is useful for the application which uses the **Pool** only at system initialization (startup). This is the safest version of the dynamic memory allocation, because it results in zero memory leakage, reasonable memory waste and no fragmentation.

Please take a look at the following code snippet.

Listing 1: API of the **Pool** pattern.

```
namespace k2lib
{
// namespace k2lib body

template <typename T, int SIZE>
class Pool
{
public:
  Pool();
  T *palloc();      /*!< Take (allocate) one element from the Pool. */
  void free(T *p); /*!< Give back (free) element of given argument pointer. */
  int size();       /*!< Returns count of free Pool elements. */

private:
  /* Rest of the elements... */
};

} // namespace k2lib
```

Above code shows API of the **Pool** pattern. It is quite straight-forward - one constructor that initializes internal memory *heap* and three methods `palloc`, `free` and `size` that are used to operate with the **Pool**. There is nothing hidden in mentioned methods, and their usage are commented in the code.

1.2.3 Pattern usage

Simple example that shows how to operate with the **Pool** is depicted below.

Listing 2: How to operate with the **Pool** class.

```
#include "Pool.h"

int main(void)
{
  cout << "Start of test..." << endl;

  const int SIZE = 4;                // Number of Pool elements
  k2lib::Pool<uint16_t, SIZE> pool; // Create Pool of 4 elements (uint16_t type)

  uint16_t *objects[SIZE]; /* Create pointers to hold dynamically allocated
                              elements */
  for (int i = 0; i < SIZE; i++)
  {
    // Allocate element from the Pool
    objects[i] = pool.palloc();

    // If allocation failed palloc() returns NULL
    if (objects[i] == NULL)
      cout << "Allocation error!" << endl;
  }

  // There is no more elements in the Pool, so p supposed to be NULL
  uint16_t *p = pool.palloc();
  if (p != NULL)
    cout << "Allocation error!" << endl;
```

(continues on next page)

(continued from previous page)

```
  // Free (give back to the Pool) element pointed by objects[2]
  pool.free(objects[2]);
  // Allocate element from the Pool
  objects[2] = pool.palloc();
  // If allocation failed palloc() returns NULL
  if (objects[2] == NULL)
    cout << "Allocation error!" << endl;

  // Test whether there are elements in the Pool (all supposed to be reserved)
  if (pool.size() != 0)
    cout << "Allocation error!" << endl;

  // Free all elements
  for (int i = 0; i < SIZE; i++)
  {
    pool.free(objects[i]);
  }

  // Test whether Pool has SIZE elements after freeing all elements
  if (pool.size() != SIZE)
    cout << "Allocation error!" << endl;

  cout << "End of test..." << endl;
  return 1;
}
```

Above example is simple and provides clear overview how to work with the **Pool** pattern. However it is not clear yet what **Pool** pattern guts look like. First of all it requires some elements storage `elements[SIZE]` and some object that keeps track of each elements status (free or allocated) - `info`. In the small microcontrollers world this `info` object could be a huge overhead and for that reason it has to be implemented in a optimized fashion.

1.2.4 Elements status tracking

My personal solution is to use bits to hold information about free/allocated elements. This approach requires creation of an array of e.g. `uint8_t` type which can hold enough bits for **Pool** size `SIZE`. This can be calculated in the following way.

Listing 3: How to calculate number of `uint8_t` elements to hold `SIZE` bits.

```
const int SIZE = 21; // Pool size (number of Pool elements) - random value only
                     // for this example.

// Number of bits in uint8_t type.
static const size_t BITS_IN_UINT8 = 8;

// Calculate number of uint8_t elements that can accomodate SIZE bits.
static const size_t NO_BYTES = (SIZE + BITS_IN_UINT8 - 1) / BITS_IN_UINT8;

// Create array of uint8_t elements of NO_BYTES size.
uint8_t info[NO_BYTES];
```

If you do calculation for yourself you should get `NO_BYTES` equal to `3` (remember of integer casting - round down).

In order to operate with the `info` array - set, clear and test bits, you can define following methods `setBit`, `clrBit`

and `testBit`.

Listing 4: Set/clear/test bits in an array.

```
void setBit(int bit_index)
{
  if (bit_index >= SIZE)
    return;

  int byte_offset = (bit_index / BITS_IN_UINT8);
  bit_index = (bit_index % BITS_IN_UINT8);
  *((uint8_t *)(&info[0] + byte_offset)) |= (uint8_t)(1 << bit_index);
}

void clrBit(int bit_index)
{
  if (bit_index >= SIZE)
    return;

  int byte_offset = (bit_index / BITS_IN_UINT8);
  bit_index = (bit_index % BITS_IN_UINT8);
  *((uint8_t *)(&info[0] + byte_offset)) &= ~((uint8_t)(1 << bit_index));
}

bool testBit(int bit_index)
{
  if (bit_index >= SIZE)
    return false;

  int byte_offset = (bit_index / BITS_IN_UINT8);
  bit_index = (bit_index % BITS_IN_UINT8);
  return (0 !=
          (*((uint8_t *)(&info[0] + byte_offset)) & (uint8_t)(1 << bit_index)));
}
```

In order to calculate position of the bit given by `bit_index` argument, you have to calculate byte position `byte_offset` in the `info` array and bit position in the target byte `bit_index`. Now, you can get `&info[0]` address and move it by `byte_offset`. Obtained `array` element can be set/cleared/tested with the calculated `bit_index - 1 << bit_index`.

1.2.5 Conclusion

There are many patterns that can help developers in memory management. Different applications require different approaches. The **Pool** pattern is one of the possible solutions. It is designed for rather applications that requires safety, then general purpose apps. It is well-suited for microcontrollers having small memory capacity. In this article you also gained a knowledge how to use bit-arrays which can be lightweight solution for e.g. `std::bitset`. I hope you have enjoyed this article and see you soon!

1.2.6 Complete implementation

In conclusion see below full code.

Listing 5: Complete **Pool** pattern implementation.

```
#ifndef POOL_H
#define POOL_H

#include <stddef.h>
#include <stdint.h>


namespace k2lib
{
// namespace k2lib body

template <typename T, int SIZE>
class Pool
{
public:
  Pool();
  T *palloc();      /*!< Take (allocate) one element from the Pool. */
  void free(T *p); /*!< Give back (free) element of given argument pointer. */
  int size();       /*!< Returns count of free Pool elements. */

private:
  T elements[SIZE]; /*!< Holds the pool objects. */

  static const size_t BITS_IN_UINT8 = 8; /*!< Number of bits in uint8_t type. */
  static const size_t NO_BYTES =
      (SIZE + BITS_IN_UINT8 - 1) /
      BITS_IN_UINT8;       /*!< Required number of bytes to hold information
                            about free/allocated pool objects. */
  uint8_t info[NO_BYTES]; /*!< Keeps track of free/allocated memory objects. */
  int free_elements_cnt;  /*!< Counts number of free elements */

  bool testBit(int bit_index);
  void setBit(int bit_index);
  void clrBit(int bit_index);
};


template <typename T, int SIZE>
Pool<T, SIZE>::Pool()
{
  // Free all pool objects
  for (unsigned int i = 0; i < NO_BYTES; i++)
    info[i] = 0xFFU;
  // Clera
  free_elements_cnt = SIZE;
}

template <typename T, int SIZE>
T *Pool<T, SIZE>::palloc()
{
  // Test whether there are any free elements
  if (free_elements_cnt <= 0)
    return NULL;

  // Look up for first pool object
```

(continues on next page)

(continued from previous page)

```
  for (int i = 0; i < SIZE; i++)
  {
    if (testBit(i))
    {
      // Found free element, so mark it as allocated and return
      clrBit(i);
      free_elements_cnt--;
      return &elements[i];
    }
  }

  // No free elements available
  return NULL;
}

template <typename T, int SIZE>
void Pool<T, SIZE>::free(T *p)
{
  // Cannot free NULL pointer
  if (NULL == p)
    return;

  // Loop through all elements and find the one to be freed
  for (int i = 0; i < SIZE; i++)
  {
    if (p == &elements[i])
    {
      // Element found, so free this element
      p = NULL;
      setBit(i);
      free_elements_cnt++;
    }
  }

  // No elements found, memory cannot be freed
  return;
}

template <typename T, int SIZE>
int Pool<T, SIZE>::size()
{
  return free_elements_cnt;
}

template <typename T, int SIZE>
bool Pool<T, SIZE>::testBit(int bit_index)
{
  if (bit_index >= SIZE)
    return false;

  int byte_offset = (bit_index / BITS_IN_UINT8);
  bit_index = (bit_index % BITS_IN_UINT8);
  return (0 !=
          (*((uint8_t *)(&info[0] + byte_offset)) & (uint8_t)(1 << bit_index)));
}

template <typename T, int SIZE>
```

(continues on next page)

(continued from previous page)

```
void Pool<T, SIZE>::setBit(int bit_index)
{
  if (bit_index >= SIZE)
    return;

  int byte_offset = (bit_index / BITS_IN_UINT8);
  bit_index = (bit_index % BITS_IN_UINT8);
  *((uint8_t *)(&info[0] + byte_offset)) |= (uint8_t)(1 << bit_index);
}

template <typename T, int SIZE>
void Pool<T, SIZE>::clrBit(int bit_index)
{
  if (bit_index >= SIZE)
    return;

  int byte_offset = (bit_index / BITS_IN_UINT8);
  bit_index = (bit_index % BITS_IN_UINT8);
  *((uint8_t *)(&info[0] + byte_offset)) &= ~((uint8_t)(1 << bit_index));
}

} // namespace k2lib

#endif /* end of include guard: POOL_H */
```

Footnote

kaeraz, 2018/11

1.3 Documentation part 1 - decomposition view

1.3.1 Background

In all my years as a programmer I participated in several huge embedded software projects. A huge project, in my view, stands for a multi-board project with many of variants e.g. board to board combinations, that development lasts over a year or two. This kind of projects involve several software teams, dedicated separately to each board, group of requirement engineers, and bunch of other people, and finally, so called managers...

Working in such project and environment:

- Requires expensive complexity management
- Requires multi-level system engineering
- Suffers from communication and synchronization issues between the teams
- Needs constant and proper contractor-customer communication
- Depends on customer product specification maturity
- Requires good **systems**, **subsystems** and **code documentation**

There are many reasons why such a project may fail, get abandoned or rejected. Most of the IT projects does not succeed in terms of time and money. However, I do not want to yammer about a project issues, my todays goal is to elaborate a bit on software development **documentation**.

1.3.2 Software architecture

At the initial project stage selected people work on **SW architecture**. There is no one definition of what **system architecture** is.

> *Architecture describes the basic organization of a system. It is embodied by its individual parts and their relations between each other and to their environment, as well as by the guidelines that manage the design and the development of the system.*
>
> —IEEE Standard 1471

In general **software architecture** is about:

- decomposing the system into parts called components, modules, elements
- defining links between the parts by the means of interfaces
  - internal interfaces (inside the developed system)
  - external interfaces (for communication with outside world and environment)
- describing SW components tasks, functions and responsibilities
- presenting components interactions in functional view

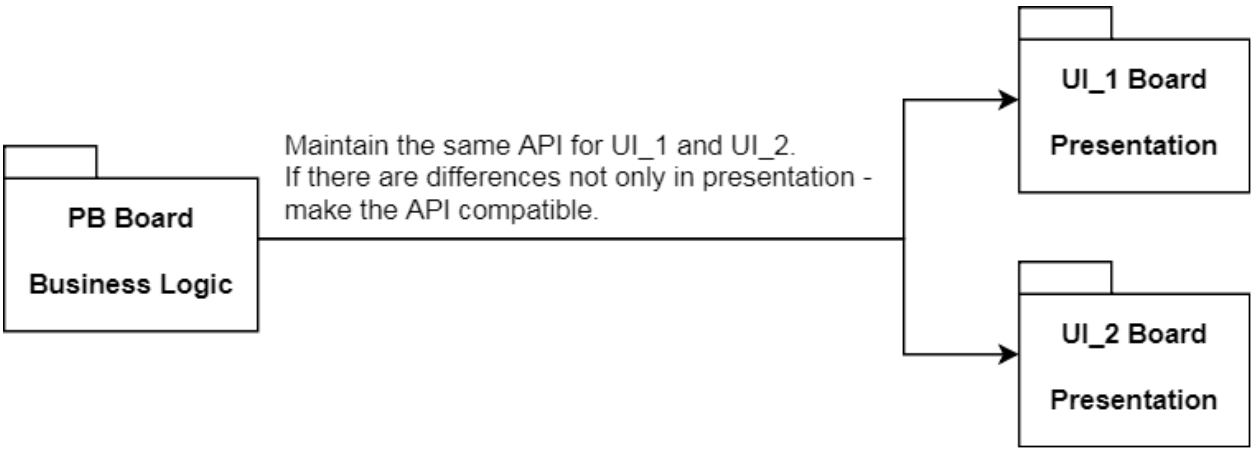
1.3.3 System overview

In this section I would like to elaborate a bit on several aspects of the system architecture's **decomposition view**. **Decomposition view** presents decomposed system - its parts, components, elements and modules. Lets discuss simple example of two boards:

- User interface board (**UI**)
- Power board (**PB**)

The **UI** board is kind of GUI that contains display and buttons. In case of washing machines **UI** board typically contains buttons, jog or rotator for program selection (that can be illuminated) and a display e.g. LCD or LED version. It is located at the top panel of the washing machine - everyone probably saw it... The **PB** board contains all drivers required for washing machine operation such as motor control driver, valves driver, heater driver, etc. By the driver I mean electronic circuitry for driving the certain actuator (motor, valve, heater).

This kind of two board solution is a typical embedded system example. Lets think a bit what both boards firmwares should look like. In my experience it is a good approach to make **UI** board totally *stupid*. It means that all the business logic and data management shall be located on the **PB** board. This solution has one huge advantage - if logic change is required, only **PB**'s firmware has to be updated and analogously if data presentation change is requested only **UI**'s firmware needs update. This is not always true because it is usually hard to completely decouple both boards' software. In the previous sentence is a small cheat. There is another big reason that stands for this solution - it is an ability to replace the **UI** board with a different one (maybe another variant, with some extra LEDs). Now, we have the same interface to both boards **UI_1** and **UI_2**, so that **UI's** firmware can only focus on data presentation.



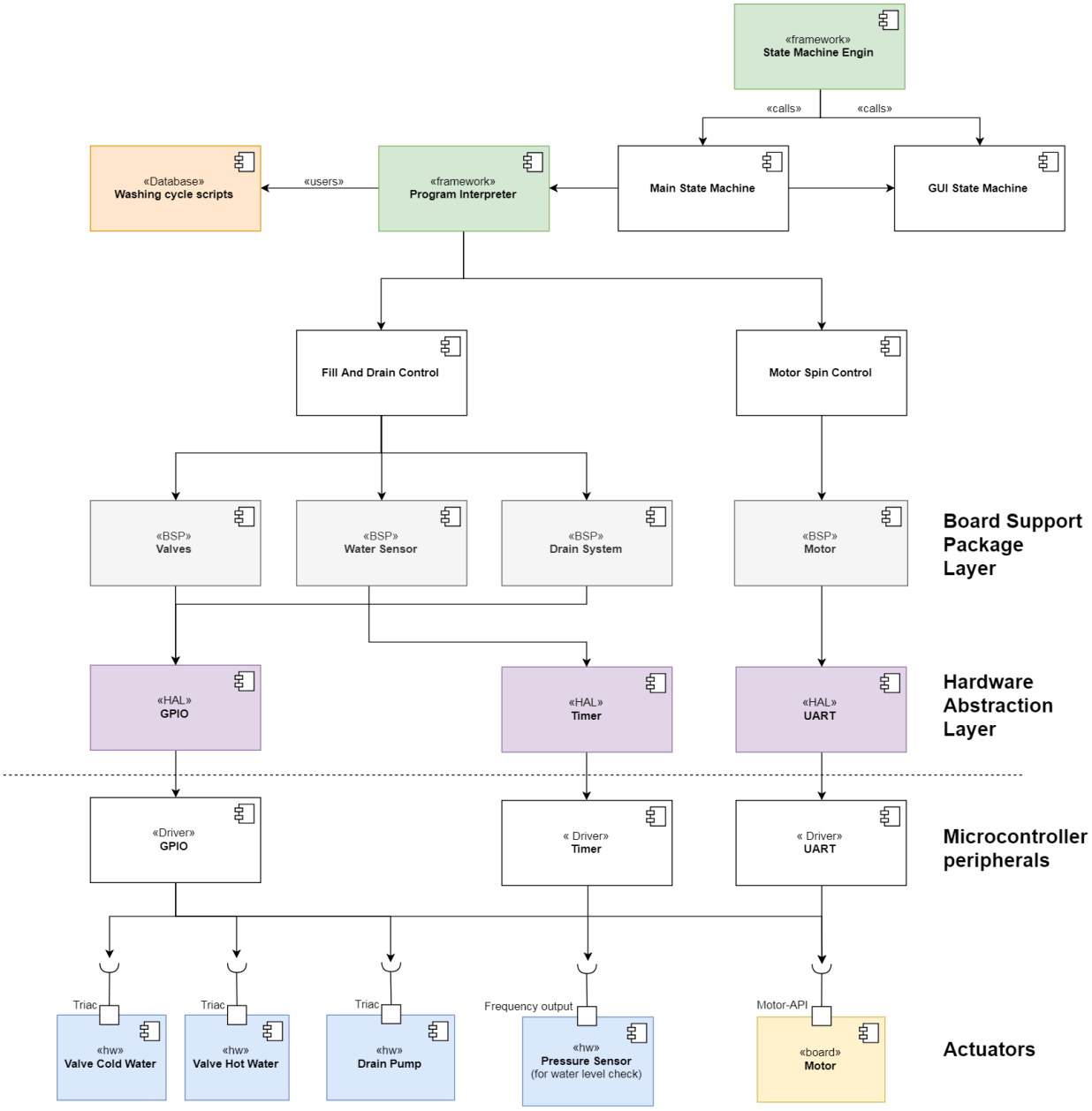
On the above picture I put a note regarding API compatibility. Lets imagine that both boards `PB` and `UI` are connected by the `UART` interface. It is reasonable to develop such `UART` protocol that is not prone to damage due to protocol API extension. This can be easily justified - lets imagine the following scenario.

1. Firmware for `PB` and `UI_1` is ready and working.
2. Now both `PB` and `UI` teams start working on new `UI` variant and lets call it `UI_2`.
3. New `UI_2` variant has extra LED 7-segment display to show time of selected washing cycle (TTE - time to end).
4. Information about the TTE has to be delivered to the `UI_2` board.

The scenario shows the case where additional information has to be delivered to `UI_2` board, in contrast to `UI_1` board which hasn't got this data. This means that protocol has to be extended by e.g. adding new message that conveys the TTE number. At this point it is not a good idea to change firmware for first variant `UI_1` - it just has to work properly when it receives TTE message from `PB`. Remember both `UI_1` and `UI_2` boards can used interchangeably with the same `PB` board and the `PB` board is configured in a way that it knows the variant of `UI` board to work with.

1.3.4 Decomposition view

OK, lets come back to the topic. The **decomposition view** shows modules and components structure and features the links (interfaces) between those elements.



The example shows typical software **decomposition view**. Each component can be treated as a `.c/.h` files (in C language) or as a group of those files that expose certain API. This API can be described in the another view - **class diagram**. The example features following:

- Components at the bottom represents electrical actuators and circuitries that expose steering interface e.g. `triac` gate input.
- Upper layer consists of `Microcontroller Peripheral Drivers` (this layer is usually provided by the microcontroller's vendor like **ST** or **TI**) and can be easily configured by provided tools e.g. **CubeMX** from **ST Microelectronics**.
- The `Hardware Abstraction layer` (`HAL`) is a layer that abstracts microcontroller's peripherals by exposing stable/frozen API to the application upper layers. As I said this frozen interface (bunch of predefined functions e.g. for controlling microcontroller's `GPIOs`) shall not change when we replace the microcontroller from e.g. **STM32** to **MSP432**. The only change required is to update lower `Microcontroller Peripheral Drivers` layer.

- One layer above the `HAL` we can find `Board Support Package` layer. It is usually optional, however quite useful in many cases. The best example of this layer we can find in the popular **dev-kits** e.g. **Arduino**, **STM32 Discovery** or **MSP430 Launchpad**. Vendor provided examples for those boards contain predefined names for the existing hardware elements e.g. **LEDs** or **Buttons**. Take a closer look at the **Arduino** among the whole range of different **dev-boards** - they varies in hardware, while software developers use mnemonic `1`, `2`, `...` for digital pins and `A0`, `A1`, `...` for analog pins. The developer does not know which microcontroller's port or pin is hidden behind the mnemonics. This makes it very easy for the vendor to produce various **dev-kits**, and developers are able to run the same software on it (no change are required - or small adjustments). Configuration of the board is done by the vendor in `Board Support Package` layer.
- The *green* components represent `Framework` modules. `Framework` is a layer that provides elements that can be used among the software layers (usually from `HAL` up). Usually `Framework` layer contains of global components (Utilities/Libraries). In the example I put `Program Interpreter` and `State Machine Engine`. The `State Machine Engine` component can be treated as a scaffolding for underlaying state machines.
- The `Washing cycle scripts` component is a database that holds washing cycles definition encoded in a custom scripting language. The scripts are used and executed by the `Program Interpreter`, which understands the commands and delegates its execution to the `xxx Control` components (low level algorithms and actions).
- The rest of the components resides in a so called `Business Logic` layer. Those modules implement the program logic (e.g. state machines for controlling the device and GUI - both communicated with each other).

1.3.5 Conclusion

This article defines **software architecture** definition and provides an example of its fundamental view which is **decomposition view**. The decomposition view is used to show software components and interfaces that link them together. The interface indicates what API is exposed by the particular component. The **decomposition view** is supplemented by the textual components description. This description should give an overview what tasks and responsibilities belongs to the components. When we develop the **decomposition view** is it crucial to know the interfaces between the components. It gives you the ability to correctly find missing modules or to decide whether to shift some component's tasks to other component. In the next article I would like to present you a way how to describe the interface and how to utilize it when defining the software's **functional view**.

Footnote

kaeraz, 2018/11

1.4 Documentation part 2 - interfaces and functional view

1.4.1 Step back

Last time I explained you software **decomposition view** which is one of many views of a **software architecture**, that you are likely to use when documenting you system. The **decomposition view** enables the designer to look deeply into a software system and break it down into software modules and components. I owe you an explanation what the component is. Please see below a texbook definition.

A **component**:

- is part of software developed for re-use
- supports information hiding
- its functionality is exposed via **interfaces**

- implements certain functionality
- can consist of many components (composite component)

In practice,

> *component is a group of source files that implements some meaningful (in terms of a system) functionality, that can be abstracted as a software part. Other software parts can operate with it through known group of methods - component's interface. This software part can be, of course, composed of many other software parts that work together and communicate with each other in defined way. We can say that a component implements some part of the software system and can be analyzed separately from other software parts.*

1.4.2 Interfaces

The concept of an **interface** enables the designers to connect system parts and understand how they communicate with each other. In other words, it shows what the component does and what doesn't.

To be clear an precise, an **interface**:

- is a list of inputs (e.g. methods) that tell the component to process (do something) and give back an output (result)
- separates the declaration (which shows what things component does) from implementation (how component does the thing)
- is a contract that component supports defined interface methods
- is a black-box

The above description is a bit confusing. To put it as simply as possible,

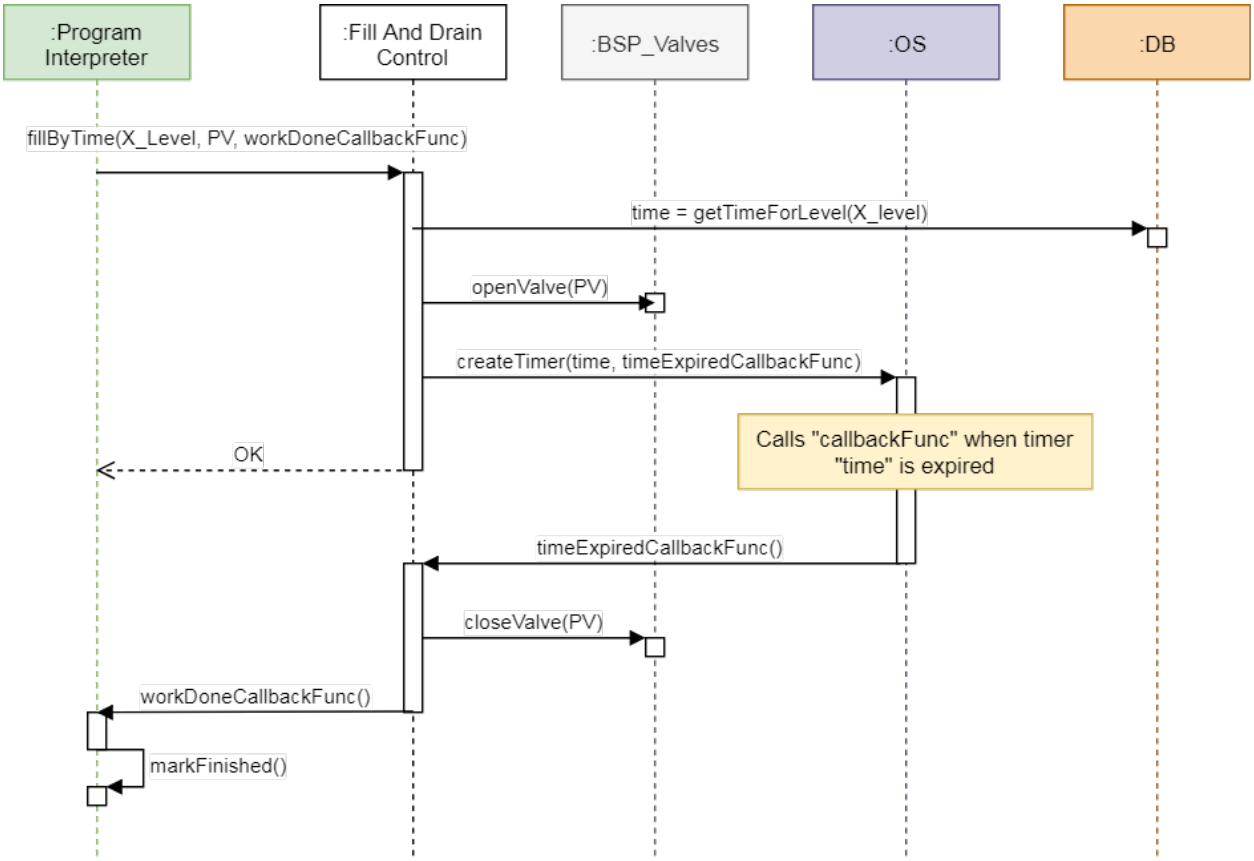
> *an interface is a list of functions, methods, operations. The component which has such an interface obligates itself to implement it. By the means of the interface you are able to invoke some component's code and get its result.*

1.4.3 Functional view

It's time to discuss how interfaces and components are utilized in practice. Knowing the ahead of time module's interface has many advantages, however it is not always possible to realize. The modeling such as software designing might be a tool that can help you to examine modules communication and corresponding interfaces. Prior to modeling, you should prepare a list of use cases or functionalities that you might want to achieve by the software. Start with a big picture and break it down into smaller scenarios. Below you can find a list of exemplary functionaries in connection to filling the water into a washing machine.

- Filling shall be executed to the defined level
- It is possible to select water inlet (valve) for different program steps
- Filling shall be executed by the time manner
- Use pressure water level sensor to check water level
- Monitor water level while filling to prevent overfilling
- etc.

Below, I would like to present realization of the functionalities on a **sequence diagram**, pointing the usage of modules' interfaces. Doing this on paper actually helped me to investigate what the interfaces should look like. The **sequence diagram** is a tool which shows particular communication scenario and features links and interactions between the software modules. Set of such dynamic diagrams forms a architecture **functional view**. The **functional view** focuses on modules communication to realize particular function of a system.



The example shows modules interactions to realize **time water filling**.

1. The `Program Interpreter` module invokes `fillByTime()` method passing tree arguments:

a. `X_Level` - water level time parameter.

b. `PV` - target value - here `PV` stands for *Prewash Valve* (used during prewash phase of the washing cycle).

c. `workDoneCallbackFunc` - callback method, which shall be called by, the `Fill And Drain Control` module when its task is finished.

2. The `Fill And Drain Control` module:

a. Asks `DB` (database) module for concrete filling time, defined by `X_Level` argument.

b. Calls `openValue()` method to open a *Prewash Valve*.

c. Creates a software timer, for the given filling time `time`, and registers a callback method `timeExpiredCallbackFunc`, which is called by the `OS` module, when the time is expired.

3. The `OS` module counts timer's time and calls a `timeExpiredCallbackFunc()` method, when the time is expired.

4. The `Fill And Drain Control` module:

a. Closes the *Prewash Valve* by calling `closeValue()` method.

b. The `workDoneCallbackFunc()` callback is called back to indicate that work is done.

5. The `Program Interpreter` marks its task as done by calling its own interface `markFinished()` method.

Here you can find modules' tasks and responsibilities that are valid for the example.

Module	Tasks and responsibilities
`Program Interpreter`	• Executes program cycle. • Invokes algorithms.
`Fill And Drain Control`	• Executes filling algorithms (e.g. filling by time). • Provides interface for high level actions related with filling.
`BSP Valves`	• Provides API for opening and closing the valves.
`OS`	• Runs the tasks. • Provides interface to manage time (Timer).
`DB`	• Stores program's configuration (e.g. filling times).

Other software modules like `HAL` and `Driver` modules, are not shown on the example for simplicity. This is your decision what level shall the **sequence diagram** keep. In general, please do not be to precise. Some details are just clear for the developers.

1.4.4 Conclusion

Software modules shall implement certain functionalities. Separation of the functionalities into modules helps to keep the system decoupled. Each module should realize only limited tasks and for that reason developers should keep in mind **Single Responsibility Principle**. Having a software system broken down into components, you are able to show modules' interactions on a so called **functional view**. The **sequence diagram** can be used for that purpose. Bear in mind, that prior to modeling **sequence diagrams**, you may not be able to break down a software system into parts. This is normal and do not blame yourself for being undecided. Use all architectural views and diagrams to investigate the system. Take your time to try many approaches and after getting enough experience, you will be sure which approach suits best for you.

Footnote

kaeraz, 2018/12

1.5 Documentation part 3 - state machine

1.5.1 Topic introduction

Nature of a typical software developer is fickle. One starts a project, and in a big hurry rushes to code. The code grows and when contain a thousand of lines it becomes obvious that it may be worth to care about its documentation. A documentation is uncommon and festive, like a salary, once in a month, and usually misused. Lots of engineers suffers from missing documentation - they who develop needs to share with their coworkers, while they who maintain the product have to worry about the code quality and consistency. But, how can one be consistent if nobody instructed him why this particular functionality has that shape? How one can extend the functionality having only its intuition of what the current implementation is. Correct documentation can be a solution.

1.5.2 Documentation

Project may have several documentation documents, where the following are the common ones.

1. **Customer Specification**

- Shows customer's point of view.
- Describes customer's wishes.
- Usually not consistent and incomplete.
- Is maintained by the customer.

2. **Product Description**

- Shows the contractor's understanding of the **Customer Specification**
- Is complete and concise
- Is enough and shows what is the subject of the contract
- Is a base for the other contractor's documents
- Is maintained by the contractor with support and consent of the customer

3. **Software Architecture**

- Designs a system
- Goes in accordance with **Product Description**
- Is maintained by the contractor
- Is a guideline for developers
- Has rules how to do the Product
- Ensures the usefullness of chosen structure

1.5.3 Missing design view

In the previous article parts we went through various of methods how to describe software system. The starting point was a **Decomposition view**, then we move to a **Functional view**, and today we are going to discuss a **State view**.

A **State view**,

- shows behavior of a system using finite state transitions
- can be described by a **State machine** UML diagram
- can be applied to any software level from software part to software subsystem
- usefull for systems that contains a state

There are many ways how to document state transitions however a **State machine** UML diagram seems to be resonable to use.

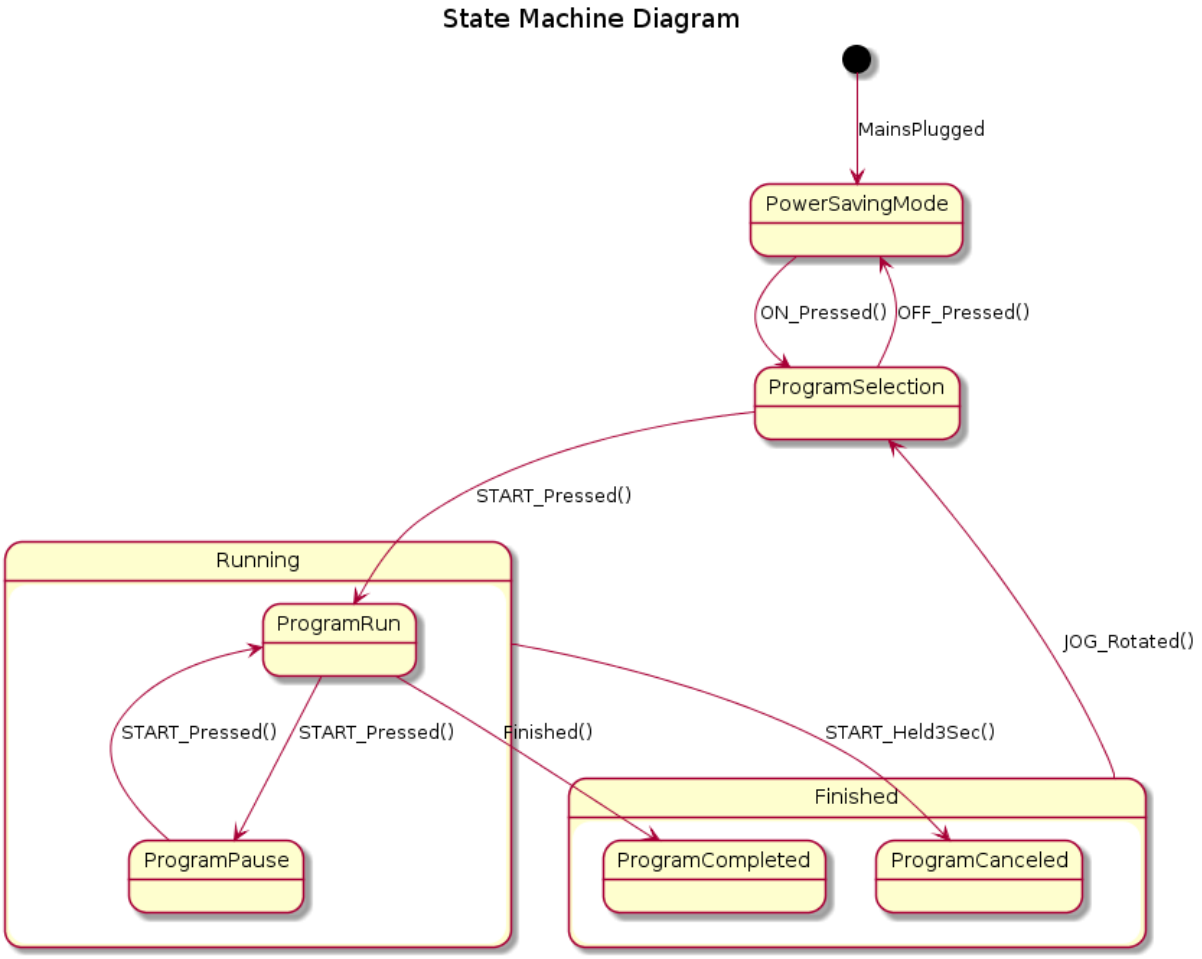
1.5.4 State Machine

When we are familiar with a litte of the teory, it is a time to show the **State view == State machine** diagram in practical example. Let's suppose that we have a washing machine composed of two boards.

- Power Board that operates the drivers and activators, and a

- UI board responsible for showing the menu for the user.

The following diagram shows a **State machine** of a Power Board operation.



It needs a little bit of exmpanation. The **State machine** consists of several states: `PowerSavingMode`, `ProgramSelection`, `Running`, `Finished`. The `Running` and `Finished` states have of course internal states: `ProgramRun`, `ProgramPause`, `ProgramCompleted`, `ProgramCanceled`.

Al the states are connected by arrows indicating state transitions on given action (trigger). In the diagram all triggers are beside the transition arrows. Example shows several triggers that some of them are closely connected with the UI boards: `ON_Pressed()`, `OFF_Pressed()`, `JOG_Rotated()`, `START_Pressed()`, `START_Held3Sec()`, it has also Power Board internal triggers like `Finished()` and environment triggers like `MainsPluggerd`.

This is not the place to descibe how to draw **State machine** diagrams. My intention is to provide you a hint that this kind of driagrams can be helpful in software engineering. The exemplary **State machine** can be bundled with a User Interface **menu diagrams**, so that user and developers can have a feeling what system options are available to the user.

A **menu diagrams** shows what the screen looks like. It also depicts the transitions between the menu screens on different user triggers (like **State Machine**). Let's take a look at the following situation.

Power Board state	UI menu
`PowerSavingMode`	• Screen is black. • Reacts only to `OFF_Pressed` trigger.
`ProgramSelection`	• Shows program selection screen. • User can select in example Spin Speed, Temp., Start Delay, etc. • The `START_Pressed` trigger changes the Power Board's state.
`ProgramRun`	• Shows progress screen e.g. Time To End value. • User can cancel the program by holding START button (emits `START_Held3Sec` trigger). • User can pause the program by `START_Pressed` trigger.
etc...	etc...

It is quite handy to put a UI menu screen picture beside each Power Board's state.

1.5.5 Conclusion

The **State Machine** diagram is a powerful tool to show stateful representation of a system. It is particulary convenient in situations where two or more systems have its own states and they have to communicate with each other in some way. My personal feeling is that simple diagram is more valuable than a stack of text.

Footnote

kaeraz, 2019/02

1.6 D2D - Introduction

1.6.1 Introduction

Devices to Devices (**D2D**) is a library for communication from **Devices** to **Devices**. It uses single write hardware connection that reduces amount cables. The library supports a Mutli-Point (Mutli-Master) topology and it is easy to used with different microcontrollers. The application protocol can be defined, as it is not imposed on the user. The Physical Layer implementation depends completely on the uses, so that the library may be used to transmit data by serial interfaces and wireless interfaces as well.

It is intended for the embedded systems like:

- IoT devices
- Various types of robots
- Home Appliance devices (Washers, Dryers, Fridges, ..)
- etc.

that contain or interface several Nodes.

The main features of the **D2D** are:

- Portability (written in simple C)
- Multi-master (Multi-point) solution (any device can send to any device with one common cable for all)
- Reduction of hardware connections (simple transmission and reception circuit)
- Supports frames collision and handling
- Layered design easy to plug-in on each level

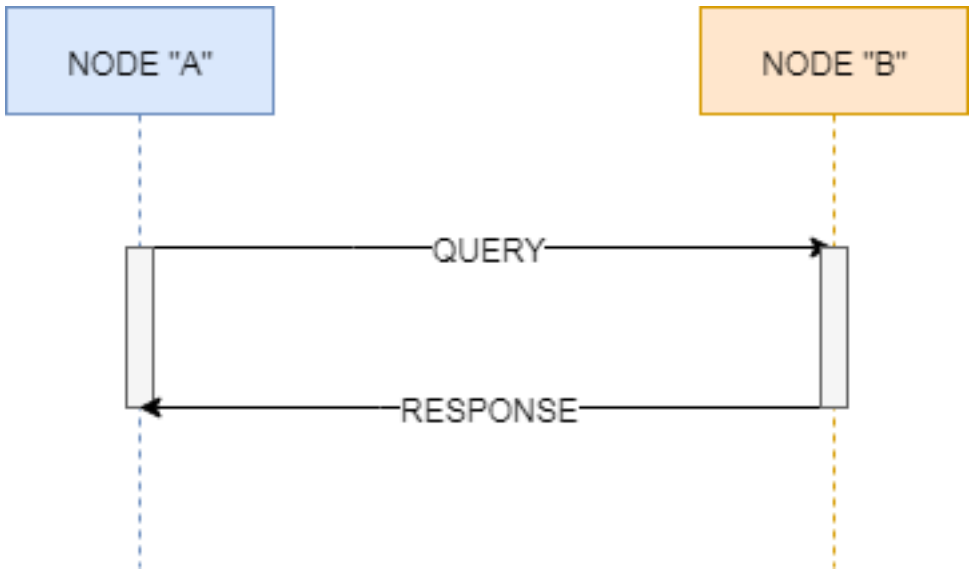
Note: The **D2D** library can be used for the interfaces that supports **peer-to-peer** model on the **Physical Layer**. The addressing is done inside the library within frame header.

1.6.2 Communication scheme

The **D2D** library assumes the request and response communication scheme. **Request** is called **Query** and **Response** is just called **Response**. The communication scenario is depicted on the below picture.

Note:

- Request message is called **Query**.
- Response message is called **Response** (to the Query).

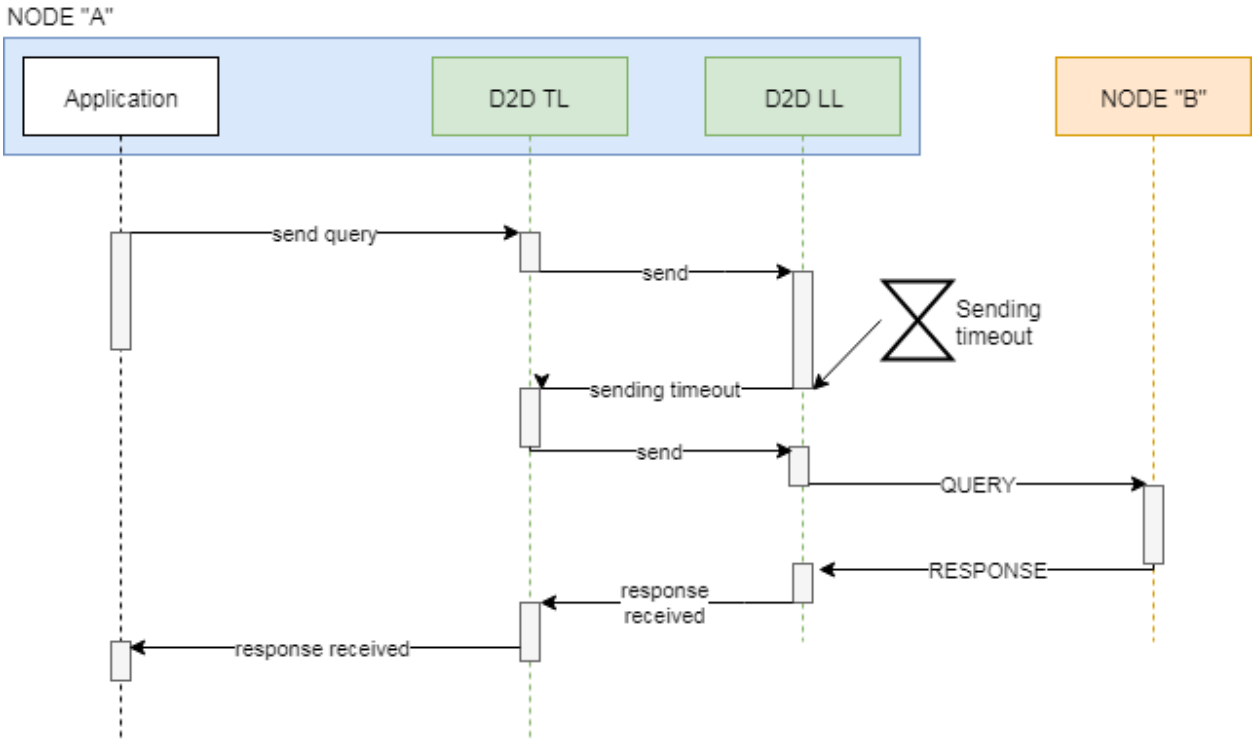


The Node `A` sends a **Query** to the Node `B` and Node `B` responds to Node `A` with a **Response**.

In case of a **Query** sending failure, the seneder re-transmits the **Query** to the recipient. Number of the re-transmissions is settable in the library. There are two types of sending failure:

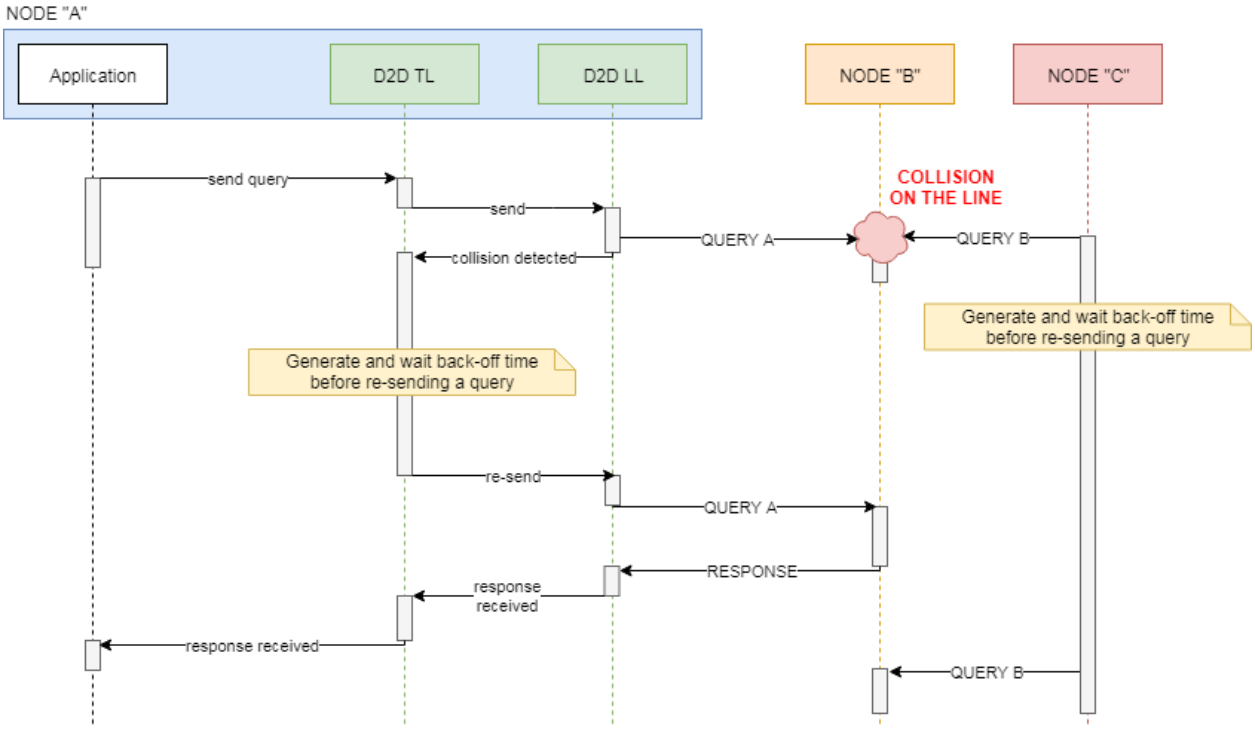
- Sending timeout
- (Optional) Sending collision

Sending timeout happens when after giving a send request to the library, the frame cannot be sent because of e.g. lack of peripheral resources. In that case the **Link Layer** informs upper layers about such event and re-transmission is performed. The re-transmission is executed immediatelly after timeout expiration.



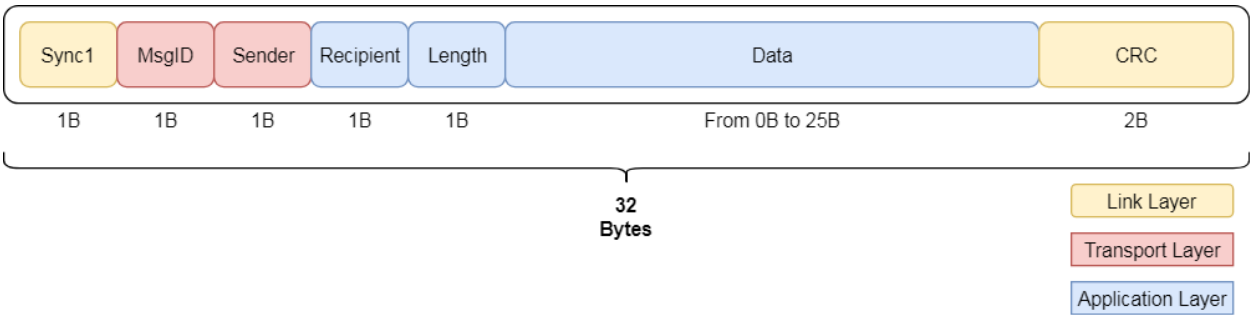
Sending collision describes a case when **Link Layer** is informed about a collision during sending the data over the medium. In **UART** example it is possible, because the transmitted signal (TX line) goes back in a loop to the sender (RX line). This echo signal is used to compare transmitted and received byte. If collision happens the **Link Layer** informs upper layers about such event and re-transmission is performed. Re-transmission is executed after some time which is called a back-off time. The back-off time is generated randomly, so that it reduces possibility of next collision.

Note: In contrary to the **Query**, there is no **Response** re-transmission. It implies that when the **Response** was not sent properly by some Node, it waits for the **Query** sender to re-transmit the **Query**. When **Query** arrives the receiving Node has a chance to send a **Response** once again.



1.6.3 Messages format

The **Query** and **Response** messages have common format which is shown below. There is no differentiation between a **Query** and **Response**. The **Response** data content depends completely on the user, so that the user may decide to carry some useful information or send just zero-length data as a **Response**. The zero-length response may be assumed as an acknowledge. In genral, when the **Response** does not come back, it means that the sender has to re-transmit the **Query**.



The colored blocks tells what **D2D** layers are responsible for what data parts. The message contains following fields:

- `Sync1` - Frame synchronization byte (**0xAE**)
- `MsgId` - Message identification number. The **MsgId** is the same for the **Query** and **Response**.
- `Sender` - Sender Node address.
- `Recipient` - Recipient Node address.
- `Length` - Number of `Data` bytes.
- `Data` - User data bytes.
- `CRC` - Checksum value that prevents from receiving distorted frames.

The `MsgId` field play an important part in the messaging strategy. All the **Responses** and all the **Query** re-transmitions shall carry the same `MsgId` value. The only exception is when one of the Nodes had an internal issue during the valid communication.

1.6.4 Messaging suggestions

In a **Query** recipient Node point of view, if receives a valid **Query** and responds to it with a **Response**, it shall wait at least one **re-transmission timeout**, and after that time it can sure that the **Query** sender Node received its response properly.

However if a **Query** sender Node during receiving a **Response** had an internal issue (e.g. HW reset), it shall send the **Query** again to the recipient Node. In that situation the **Query** recipient Node might think that the **Response** to the first **Query** had been delivered back successfully. If it receives a new **Query** it shall accept it and respond with some data.

1.6.5 What next

Feel free to see next articles that describe in details how to use D2D library.

Footnote

kaeraz, 2019/02

1.7 D2D - Hardware connection

1.7.1 Layer 1: Physical Layer overview

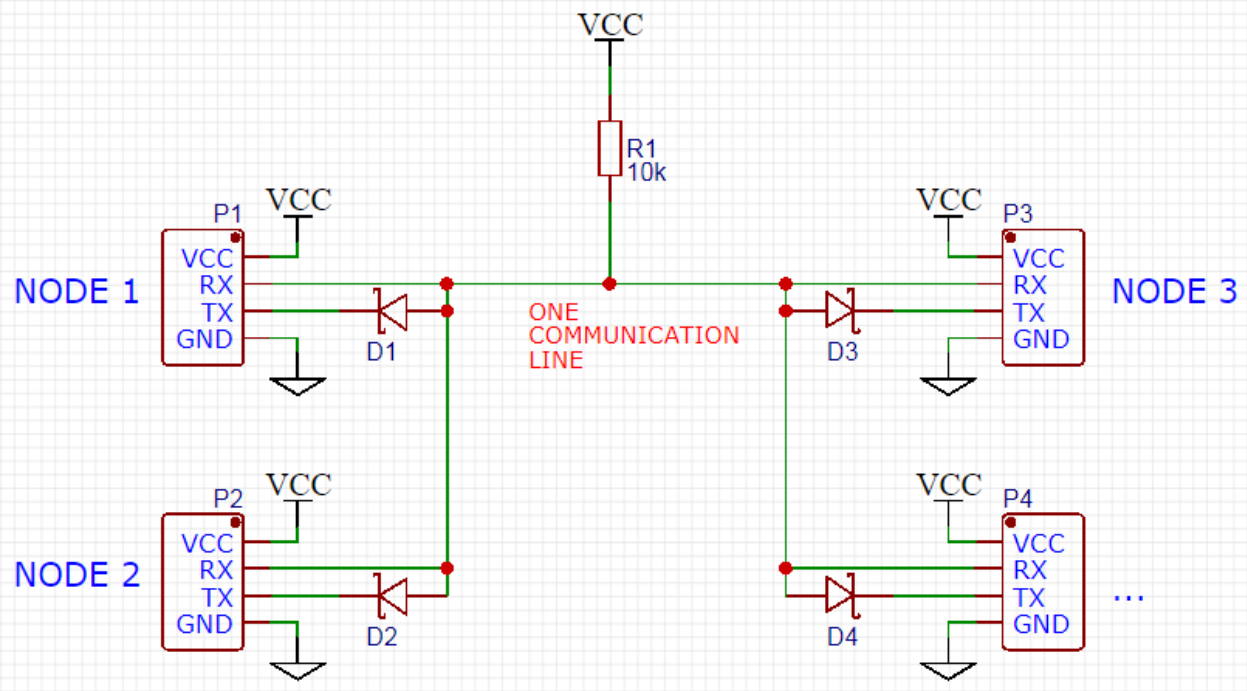
The **D2D** library provides an example of **L1** hardware connection with the use of an **UART** microcontroller interface. In general the library **Physical Layer** is responsible for:

- Sending bytes over the medium
- Receiving bytes over the medium
- (Optional) Collision detection
- Medium interface error detection
- Frame CRC calculation and verification

Note: Examplary code and hardware is based on UART microcontroller's peripheral.

1.7.2 Hardware circuit

Example solution of **L1** uses standard **UART interface** for sending signals between the devices. This decision gives simplicity and enables easy portability of the software for different microcontrollers. The simpliest hardware connection between the Nodes is shown below. It uses standard **UART** interface with **RX** and **TX** lines.



The **Physical Layer** implementation depends completely on the use cases, so that one can utilize different transmission medium like wireless or wire serial communication.

1.7.3 What next

Go to the next article to find out how to setup **D2D** library.

Footnote

kaeraz, 2019/02

1.8 D2D library setup

1.8.1 Introduction

This chapter describes how to seput the **D2D** library. In general, correct setup requires few actions:

- setup of **Link Layer** to support used microcontroller
- **Pseudo Random Number Generator** setup
- **Timer** module setup

Note: The **D2D** library depends on two additional modules: **Timer** module and **PSNG** module.

1.8.2 Layer 2: Link Layer overview

The **Link Layer** also called **Data Link Layer** is responsible for transfering data between two devices. It defines how devices gain the access to a medium. The **Link Layer** organizes data before transmission and verifies whether the data

were received or sent successully. The best example of it could be a collision detection feature - which is supported by the **D2D** libray. The **L2** plays an important role in error detection and error recovering. In case of an **UART** examplary implementation, **Link Layer** can detect medium transmission errors such as **parity** or **overrun**, or framing erros such as **CRC** mismatch.

1.8.3 How to setup

After building a basic hardware connection between the Devices (Nodes), you have to customize the **Link Layer** for the target microcontroller. In order to do this, you have to take a look at the below files.

```
d2d_ll_config.h
d2d_ll_config.c
```

The `d2d_ll_config.h` file contains an API that is used internally by the library. The only thing you have to do is to provide its implementation inside the `d2d_ll_config.c` file.

The implementation determines what options are supported by the library (e.g. collision detection). It also enables the user to port the code to different platforms and use various transmission medium.

Let's take a look at the API of the `d2d_ll_config.h` file.

```
void d2d_ll_peripheral_init(void);
void d2d_ll_peripheral_abort_transmission(void);
void d2d_ll_peripheral_enable_rx_interrupt(void);
void d2d_ll_peripheral_disable_rx_interrupt(void);
void d2d_ll_peripheral_enable_tx_interrupt(void);
void d2d_ll_peripheral_disable_tx_interrupt(void);
void d2d_ll_peripheral_transmit_byte(uint8_t data);
uint8_t d2d_ll_peripheral_get_received_byte(void);
```

Refer to the **Doxygen** documentation of the module to see how to implement the above functions. Below I will describe the default **UART** `d2d_ll_config.h` API implementation.

1.8.4 Default UART implementation

The default implementation uses **UART** microcontroller's peripheral with interrupt mode. The are two interrupts supported: transmission and reception. Sometimes both interrupts are enabled simultaneuslly so that it is possible to detect the collision when sending - this feature is optional.

Below examplary code uses **Low Level (LL)** library provided by **ST Microelectronis** for **STM32** microcontrollers.

```
void d2d_ll_peripheral_init(void)
{
    // Peripheral is initialized outside by the CubeMX tool
}

void d2d_ll_peripheral_abort_transmission(void)
{
    // Disable both RX and TX interrupts
    d2d_ll_peripheral_disable_rx_interrupt();
    d2d_ll_peripheral_disable_tx_interrupt();

    // Get byte to clear the register
    (void)d2d_ll_peripheral_get_received_byte();

    // Clear ERROR flags
```

(continues on next page)

(continued from previous page)

```
    LL_USART_ClearFlag_PE(USART1);
    LL_USART_ClearFlag_FE(USART1);
    LL_USART_ClearFlag_NE(USART1);
    LL_USART_ClearFlag_ORE(USART1);
    LL_USART_ClearFlag_IDLE(USART1);
    LL_USART_ClearFlag_TC(USART1);
    LL_USART_ClearFlag_LBD(USART1);
    LL_USART_ClearFlag_nCTS(USART1);
    LL_USART_ClearFlag_RTO(USART1);
    LL_USART_ClearFlag_EOB(USART1);
    LL_USART_ClearFlag_CM(USART1);
    LL_USART_ClearFlag_WKUP(USART1);

    // Get byte to clear the register again
    (void)d2d_ll_peripheral_get_received_byte();
}

void d2d_ll_peripheral_enable_rx_interrupt(void)
{
    // Reception interrupt enable
    LL_USART_EnableIT_RXNE(USART1);
}

void d2d_ll_peripheral_disable_rx_interrupt(void)
{
    // Reception interrupt disable
    LL_USART_DisableIT_RXNE(USART1);
}

void d2d_ll_peripheral_enable_tx_interrupt(void)
{
    // Transmission finished singal for library code
    // is done by enabling Reception interrupt.
    // This configuration supports the collision detection.
    // After the library sends the data byte, the same
    // byte supposed to come back to the UART and be
    // received, so that it triggers the RX interrupt.

    // Library additionally counts the sending timeout,
    // what prevents from stacking in waiting for interrupt
    // state.
    LL_USART_EnableIT_RXNE(USART1);
}

void d2d_ll_peripheral_disable_tx_interrupt(void)
{
    // Reception interrupt disable (see description
    // for d2d_ll_peripheral_enable_tx_interrupt() function
    // to see why RX interrupt is disabled here.
    LL_USART_DisableIT_RXNE(USART1);
}

void d2d_ll_peripheral_transmit_byte(uint8_t data)
{
    // Set transmission data register with data, what triggers
    // byte send.
    LL_USART_TransmitData8(USART1, data);
```

(continues on next page)

(continued from previous page)

```
}

uint8_t d2d_ll_peripheral_get_received_byte(void)
{
    // Get reception data register
    return (uint8_t)LL_USART_ReceiveData8(USART1);
}
```

Above code is full of comments giving full overview how to implement the API for different microcontrollers.

1.8.5 Layer 4: Tansport Layer overview

The **Transport Layer** is a layer 4 in the **OSI** model. It is reponsible for providing end-to-end reliable communication between two logical Nodes. This layer ensures the user that data sent between the devices are correct and error-free. The **L4** manages a traffic over the medium. It controls a bandwidth by imposing speed reductions, controls message re-transmissions in case of framing errors and enusers data integrity such as maching a **Response** to the **Query**.

1.8.6 How to setup

There is no need to setup the **Transport Layer** and after it's initialization it is ready to go. However, there is one module on which the **L4** depends on. It is a **Pseudo Random Number Generator** (**PRNG**) module.

1.8.7 Dependencies: Pseudo Random Number Generator setup

The **Pseudo Random Number Generator** module is used by the **Transport Layer** in order to generate random back-off times. It requires a small setup for a target microcontroller. Required setup can be done inside a `prng_config.c/h` files. Please take a look at the `prng_config.h` API.

```
void prng_init_bit_generator(void);
void prng_start_bit_generator(void);
void prng_stop_bit_generator(void);
uint16_t prng_bit_generator_get(void);
int prng_delay(void);
void prng_seed(unsigned int x);
```

Each function contains a descriptive doxygen comment. It is probably more interesting how to provide quick but reliable implementation of the above functions.

The random bit generation can utilize an **ADC** microcontroller's peripheral. The least siginificant bit of an **ADC** conversion are usually unstable and floating. The test and not production code may use the analog input not connected, but just floating. Please refer to an examplary implementation of a **Pseudo Random Number Generator** in the **D2D** repository. The quick-and-dirty implementation for **ST Microelectronis STM32** microcontrollers is shown below.

```
void prng_init_bit_generator(void)
{
    // ADC peripheral is done outside by the CubeMX Tool.
}

void prng_start_bit_generator(void)
{
    // Start the continuous ADC measurement
    HAL_ADC_Start(&hadc2);
```

(continues on next page)

(continued from previous page)

```
}

void prng_stop_bit_generator(void)
{
    // Stop the continuous ADC measurement
    HAL_ADC_Stop(&hadc2);
}

uint16_t prng_bit_generator_get(void)
{
    // Get the lest siginificant bit of current ADC value
    return ((uint16_t)(HAL_ADC_GetValue(&hadc2) & 0x0001));
}


int __attribute__((optimize("O0"))) prng_delay(void)
{
    int data = 0;

    // Wait some time by looping. This delay is used
    // by the PRNG module to wait between consecutive
    // bit readings.
    for (int i = 0; i < 1000; i++)
    {
        data += i;
    }

    return data;
}


void prng_seed(unsigned int x)
{
    // Initialzie the pseudo random number generator
    // with a given seed.
    srand((unsigned int)x);
}

uint16_t prng_rand(uint16_t min, uint16_t max)
{
    uint16_t randomNumber;

    // Return the random number in a range
    // from min to max.
    randomNumber = (rand() % (max - min)) + min;

    return randomNumber;
}
```

The next section describes the **Timer** modules.

1.8.8 Dependencies: Timer setup

The **D2D** library depends on **Timer** module, which provides a time counting facility. When **Timer** object is created it registers user callback method that is called after time expiration. There is little to setup in case of **Timer** module. One has to set a maximum number of **Timer** objects in the `timer_config.h` file.

```
/**
 * @brief Defines number of Timers used in the application.
 *
 */
#define TIMER_NO_TIMERS  (2) // Value of 2 is a minimum number of Timers
                             // for D2D library.
```

The **D2D** library requires at least two **Timer** objects for proper operation.

1.8.9 What next

After having an overview how to setup a **D2D** library, there is a time to dig into an example code. The next article covers this topic.

Footnote

kaeraz, 2019/02