Wiki »

AMiRo-OS

This real-time operating system for the MCU based modules of AMiRo is based on ChibiOS/RT and comprises multiple drivers for peripheral hardware as well as an abstraction layer to ease application development.
For instructions how to setup your environment, execute the code on a robot, and to develop new applications, please refer to the `README.txt` file provided in the repository.

The remainder of this wiki page provides information for users and developmers, such as conventions regarding the code style and a guide on how to write efficient C++ code for microcontrollers.

User Guide

This guide provides information about how to interface AMiRo in general and which commands are the most important.

Accessing AMiRo

AMiRo provides a simple shell that is available through the serial port, which can be accessed via the programming port.
As soon as the robot is connected to your computer using an AMiRo programming cable, it can be accessed with a serial terminal application like GTKTerm (recommended) or HTerm.
Toggling the RTS signal on and off again will make the system restart. After some information about the system is printed, the shell is available as soon as the prompt "`ch>`" appears.

Using the Shell

The most important shell command is "`help`". It will print a list of all available commands on the module. Most of these commands feature their own help text when called with no additional arguments.
Since AMiRo consists of multiple modules, the command "`shell_board <idx>`" can be used to switch to another module. `idx` is a mandatory argument to specify the module to switch to, as stated in the following table.

idx	module
1	DiWheelDrive
2	PowerManagement
3	LightRing

Developer Guide

When you want to modify the software (AMiRo-BLT or AMiRo-OS), it is highly recommended to read this guide before.
First, the code style of this project is described (it is very similar to Google style). Thereafter, some hints for efficient C++ programming with MCU target platforms are given.

Code Style

When developing new software for AMiRo-OS, please follow this style guide in order to keep the formatting of the code consistent.

Avoid Common Mistakes

- `if`-statements or loops with one line also have brackets
- initialize all variables
- initialize all pointers (at least to `NULL` (C) or `nullptr` (C++))
- do not mix pointer and non-pointer variables

```
// do not do this:
int va = 0, *pa = NULL;
// or this
int* va = 0, pa = NULL;

// this is ok
int va = 0, vb = 0;
int *pa = NULL, *pb = &vb;

// this is best
int va = 0;
int vb = 0;
int* pa = NULL;
int* pb = &vb;
```

- always check pointers for `NULL`/`nullptr` before using them
- if a pointer must never be `NULL` use a reference instead
- global constants are defined as macros (C) or as constexpression (C++)

```
#define MY_GLOBAL_CONSTANT = 3.14f

namespace nutshell
{
  constexpr float nut = 12e3;
}
```

- do not use magic numbers

```
// do not use a magic number
float fooPerSecond = float(fooPerMinute) / 60;

// define constants instead
#include <path/to/my/constants.h>
float barPerSecond = float(barPerMinute) / SECONDS_PER_MINUTE;
```

Source Code Style Convention

- filenames are written all lowercase and without underscores (`demonstrationclasscppheader.hpp`)
  try to keep filenames short for readability reasons (e.g. "`demo.hpp`")
- 120 characters per line
- indentations are two spaces, no tabs
- variables and functions are written in camelCase with initial lowercase letter ("`thisIsAVariable`" and "`thisIsAFunction()`")
- class names are written in CamelCase with initial capital letter ("`ThisIsAClass`")
- constants are written `ALL_CAPITAL` with underscores
- underscores are forbidden
  exceptions:
  - when naming constants ("`SECONDS_PER_MINUTE`")
  - use them to indicate the SI unit ("`nodeWidth_nm`")
  - in low-level code (consult your supervisor!)
- the prefix "`my`" is forbidden (e.g. "`myVariable`")
- use spaces after comma and control statements

```
for (x = 0; x < 100; ++x) {
  someFunction(a, b, c);
  ...
}
```

- place braces on same line as control statements

```
if (x == y) {
  ...
} else if (x > y) {
  ...
} else {
  ....
}
```

- use proper names for index variables

```
for (uint8_t idxRow = 0; idxRow < numRows; idxRow++) {
  for (uint8_t idxCol = 0; idxCol < numCols; idxCol++) {
    printf("Value of row %d and column %d: %d", idxRow, idxCol, jacobi[idxRow][idxCol]);
  }
}
```

- spaces in templates

```
std::vector< uint32_t > proximityRingValues(8, 0);
std::vector< std::vector< uint8_t > > somethingElse(128, std::vector< uint8_t >(128, 255));
```

Coding Conventions

- always use "`this->`" to access members
- using whole namespaces, i.e. "`using namespace std;`" in files meant to be included (such as all header files) is forbidden; use "`using namespace std;`", "`using std::cout`" or "`namespace qi = boost::spirit::qi`" inside scopes only
- try to make use of `enum`, `const` variables, return values, and functions, as well as `constexpr`, or even `__attribute__((noreturn))` (explanation) as often as possible to increase efficiency
- implicit type conversions are forbidden
  Use C++ style conversions (e.g. "`static_cast<>()`") and only use C style conversions for readability reasons.
- strictly differentiate between C (.h and .c files) and C++ (.hpp and .cpp files)
  For example, `NULL` must only be used in C files, the corresponding constant for C++ is `nullptr`.

Documentation

In general every class, function, method, variable, or constant has to be commented. Inline comments help as well. Keep comments synchronized with the actual implementation.
We use doxygen-style comments, i.e. C-style comments with two initial asterisks and the @-syntax for doxygen's special commands.

- functions

```
/**
 * @brief An example method to demonstrate a function header.
 *
 * @param[in]     constIn  A constant input value.
 * @param[in]     in       An input value that may be changed by the method.
 * @param[in,out] inAndOut A reference to a struct that is filled by the method and contains output informatio
 * @param[in]     optional A pointer to some data that may be NULL and thus is optional.
 *
 * @return An error code generated by the method.
 */
const ErrorCode exampleMethod(const int constIn, float in, SomeStruct &inAndOut, SomeData *optional = NULL);
```

- variables

```
/** @brief number of seconds in one minute */
uint32_t secondsPerMinute = 60;
```

Physical Units

- variables are named according to their physical representation

```
int32_t roundsPerMinute = 0;
```

- variables are explained at their declaration with meaning and their physical SI-unit
- *integer* values with a physical representation are implicitly multiplied by factor e-6 (µ), unless stated otherwise in the definition's comment or optionally in the variable name

```
// angular velocity in 1 µrad/s
int32_t initialAngularVelocity = 1000000;
// speed of light in m/s
uint32_t speedOfLight = 299792458;
```

- *floating-point* variables are implicitly without any factor, unless stated otherwise in the definition's comment or optionally in the variable name

```
// angular velocity in 1 rad/s
float initialAngularVelocity = 1.0f;
```

Source file header

Every file begins with the following comment (comment syntax depending on file type, of course):

```
AMiRo-OS is an operating system designed for the Autonomous Mini Robot (AMiRo) platform.
Copyright (C) 2016..2017  Thomas Schöpping et al.

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 3 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, see <http://www.gnu.org/licenses/>.
```

If the file is not part of AMiRo-OS but is associated with another AMiRo project (i.e. AMiRo-LLD or AMiRo-BLT) the first line(s) of the file header must be applied accordingly. Just copy the text from another file of the project.
If you are not directly associated with the AMiRo-OS development team, but want to publish you code yourself, you may (actually you have to) modify the second line as well.

Further information about author and purpose of the file (description) should be added in Doxygen style.
In case the file is not meant to be part of the generated documentation (e.g. bash scripts or README files), this information should be stated right after the file header

as specified above.

Efficient C++ for MCUs

This guide is meant to give some advice, how to write efficient and easy-to-understand C++ code for micro-controllers (MCUs).
It was written for advanced C++ developers, who are firm with C++ syntax and technical terms.
Before you read this guide, make sure to take a look at the style guide.

Memory

Since memory load is a crucial factor for MCUs, this is a very important topic.
The more efficient the available memory is used, the more complex the applications can be.

Three Types

In general there are three ways how memory is allocated by the system:

- **static data**
  When writing code like

  ```
  static uint32_t svar = 0;

  int main() {
    return 0;
  }
  ```

  four bytes will be statically allocated on initialization.

- **dynamic data**
  Each time `malloc()`, `calloc()`, `realloc()`, or `new()` are called, a chunk of memory is allocated dynamically at run-time on the heap.

- **temporary data**
  Temporary variables as well as arguments for function calls are stored on the stack of the thread.
  For instance, in the following code snippet at least 28 bytes will be stored on the stack.

  ```
  int32_t add(const int32_t x, const int32_t y, int32_t& result) {
    // the result of the addition is first stored on the stack before it is copied to the variable (4 bytes)
    result = x + y;
    return result;
  }

  int main() {
    // temporary variable with a size of 4 bytes
    int32_t a = 10;

    // function call with three arguments of 4 bytes each (12 bytes)
    // in C++ a pointer to the instance (this) is passed implicitly as further argument for non-static functions
    // the instruction pointer where to return after add() has been processed is stored on the stack (4 bytes)
    // the result of add() is stored on the stack before it is copied to the variable (4 bytes)
    a = add(a, a, a);

    return a;
  }
  ```

One of these is good, one is bad, and the last one is actually the first one in our case.
So let us take another look on which is which.

- **static data**
  Starting with static memory, this is the best case since it is allocated when the system is initialized.
  This enables the compiler to optimize the memory layout (e.g. aligning structures) and you do not have to handle failed dynamic allocations that return a `NULL` pointer.
  When using static data only, the memory utilization of the whole software is deterministic, thus it is possible to check whether the MCU can handle the code at compile time.

- **dynamic data**
  Dynamic memory allocation must be avoided for real-time software! There are no exceptions to this rule!
  When trying to allocate memory at runtime, you always have to handle the case that the allocation function fails and a `NULL` pointer is returned. This will blow up your code and performance will suffer from these checks.
  Furthermore, dynamic memory allocation makes the system non-deterministic regarding processing time, since it is impossible to predetermine how often (how long) allocation fails.
  People might argue that this problem could be neglected if the allocated chunks of data are very small, but don't be fooled. Like most MCU operating systems, ChibiOS does not implement a mechanism to rearrange fragmented memory, thus an allocation might fail even when there are much more bytes available than you need. Take a look at this example:
  1. There are ten bytes of available memory.
     `_ _ _ _ _ _ _ _ _ _`
  2. Eight variables of one byte each are allocated and occupy the leading eight bytes.
     `X X X X X X X X _ _`
  3. Six of these variables are freed again so that only two bytes are still occupied and eight bytes are available, respectively.
     `_ _ X _ _ _ X _ _ _`
  4. A single four byte wide variable is requested but the allocation fails even though there are twice as many bytes available. Because of the fragmented memory there is no sequence of four adjacent available bytes.

- **temporary data**
  In ChibiOS the whole stack of each thread is allocated when the object is created. Since we have learned that all objects should be initialized statically on startup, the stack memory becomes static as well.
  The advantage of this approach is obvious. No memory needs to be managed dynamically during run-time for the stack.
  The disadvantage, however, is the specification of the size of this chunk of memory by the developer through a template argument. A too small stack can result in a system failure (at run-time) but a too large stack is a waste of precious resources.

Optimizing Memory Utilization

In order to choose the most efficient way of memory utilization there are some simple rules:

- Do not allocate memory dynamically at run-time!
  If you need to store data, create a new static memory structure or use an existing one. In case the problem can not be solved this way, use a memory pool (see ChibiOS documentation).

- Allocate all memory statically on initialization!
  In other words, all memory you will ever need for your computation must be allocated before the `main()` method is called. This includes threads as well as containers, buffers, memory pools, or any other objects.

- Do not waste memory!
  - Stack sizes should be as small as possible, so they are sufficient in any possible situation but do not waste bytes that will never be used.
  - Think twice when creating a new buffer or memory pool. As with the stack, these should just be as large as required.
  - Avoid pointers/references when referring to data structures that are smaller than a pointer.

```
uint8_t  a = 0;  // 1 byte
uint8_t  b = a;  // 1 additional byte
uint8_t *c = &a; // 4 additional bytes (on 32bit architectures)
```

- Optimize stack usage for function calls!
  - In many cases the return value of a function is copied and thus the temporary result value will occupy additional memory. Especially when the returned value is larger than a pointer, it might be preferable to add a reference of the object where the result shall be stored as an additional argument to the function (*return by reference*).
  - If a function manipulates a complex data structure and no race conditions will occur, hand a reference of the data to the function so no copy is required.
  - In cases where a function should not return anything, use `__attribute__((noreturn))(` explanation).
  - Remember: in C++ when calling a member function, the pointer to the instance (`this`) is an additional implicit argument. Whenever you can access data via `this->`, do not hand it as an argument to the function.
  - That said, every function call will occupy at least 8 bytes on the stack (`this`-pointer and return address). Thus, small functions should be inlined (although the `inline` keyword is nothing more but a *hint* for the compiler).

- Do not store constant values that are known at compile-time!

```
// never do this
static const uint32_t secondsPerDay = 86400;

// this is ok but still not good
#define SECONDS_PER_DAY = 86400;

// this is the best case
namespace minute {
  constexpr uint8_t  seconds = 60;
}
namespace hour {
  constexpr uint8_t  minutes = 60;
  constexpr uint16_t seconds = minute::seconds * minutes;
}
namespace day {
  constexpr uint8_t  hours = 24;
  constexpr uint16_t seconds = hour::seconds * hours;
}
```

- Choose primitive types with care!
  If the range of possible values for a certain variable is known beforehand, choose the appropriate type. This rule also applies to enumerations.

```
uint16_t relativeValueAsOneOfThousand;
enum TenValues : uint8_t { ... };
```

- Use flag-masks instead of multiple boolean variables!
  In C/C++, a `bool` is stored as a byte, even though a single bit would suffice. If an object requires multiple boolean variables, these will occupy a lot of memory. In such cases you can use masks or bitfields for such purposes.

```
uint8_t flags;
constexpr uint8_t flagAMask = 0x01; // use like '(flags & flagAMask) ? ... : ... ;'
constexpr uint8_t flagBMask = 0x02;
constexpr uint8_t flagCMask = 0x04;

struct bitfield {
  uint16_t flagA            : 1; // access via 'bitfield.flagA'
  uint16_t flagB            : 1;
  uint16_t flagC            : 1;
  uint16_t threeValueFlag   : 2;
  uint16_t somethingStrange : 9;
```

```
  uint16_t rest             : 2;
};
```

Multiple Memory Regions

This chapter only applies to MCUs with multiple distinct memory regions.
With the AMiRo this is the case for the STM32F4 of the PowerManagement module.

The STM32F4 MCU features four distinct memory regions with a total size of 196KiB.

- The *sram1* - or just *ram* - is the largest region and can hold up to 112KiB of data.
  It is connected to both DMA controllers and thus can be seen as general purpose memory. As long as an application does not require more than 112KiB memory and parallel memory access is no bottleneck, using only this region will suffice.
- If the application is very heavy on data transmission via Ethernet, UART, or/and CAN, the access of the memory can get problematic. With the two DMA controllers and the processor, there are three systems accessing the memory. This can result in high latencies and even in data loss, if external input can not be stored before the next package arrives.
  In such cases the additional *ethram* should be used, which is rather small (16KiB) but can be accessed by all three sub-systems as well.
- As an application gets more complex, memory becomes a very precious resource. However, most of the data does not need to be available for external devices, such as the DMA controllers, but only for the processor.
  For these cases the STM32F4 features additional 64KiB core-coupled memory (*ccm*), which only the ARM core can access. However, when using this memory region, a developer always has to keep in mind that the only way to move data from and to the *ccm* is by actively copying it using the processor, which is much slower and less efficient than using DMA.
- The fourth memory region is the *backup sram*. Even though it is just 4KiB small, this region is still powered when the MCU is in standby mode.
  Obviously its originally purpose is to hold information while the system is in low-power mode, as it is done for the AMiRo-BLT startup procedure.
  Hence, it should not be used as general purpose memory for applications.

Processing

There are much less rules for efficient computation with a MCU than for the memory utilization.
In fact, for classical CPUs (x86 as well as ARM) there are more things to be aware of than for low-performance, energy efficient platforms like the STM32F1 and STM32F4.
For this reason, this guide will not start with new rules but it will point out which classical rules do not apply for MCUs and emphasizing known rules that are even more important when developing code for MCUs.

- There is no cache!
  Forget about cache-coherency and efficient memory access. There is just the (tiny) memory and the registers. Some more powerful MCUs like the STM32F4 feature capabilities to access additional external memory, which might be slower than the internal one, but still there would be no cache.
- Compute twice before you store!
  Usually, memory is the much more critical resource than processing time. For this reason it often is the better choice to compute a value anew for multiple times than to store it in memory.
  For example, when you want to know the angular velocity of the robot, you will need two measurements of the current orientation and the time difference between these two. Hence, you end up with three values so far which probably can not be optimized and will be stored in memory. In order to get to the angular velocity, you now need to compute the difference of the two orientations and then divide it by the time delta. On the one hand, you can do these last steps once and store both or one value, which will be an increase in memory usage of 33% or 66% respectively. On the other hand, you can save the memory and recompute the speed everytime you need it.
- Back to the roots!
  'Advanced' features such as floating point acceleration or SIMD are very rare in the MCU world and only available on few chips.
  For the AMiRo, there is the 'big' STM32F4 MCU, which supports hardware accelerated floating point computation. Hence, `float` data can be computed as fast as `int`. However, there is no support for `double` precision.
  The 'small' STM32F1 has no such hardware unit and thus all operations with `float` data take 2 to 12 times longer than operations with `int`.
  A possible drawback of floating point units (FPUs) is that they provide additional registers, which increase the size of the context for each thread and makes rescheduling take longer.
  However, most FPUs can access the ALU registers as well, so simply not utilizing the additional register memory keeps the context small and fast.
- Let the compiler do the work!
  Whenever a formula can be solved at compile time, let the compiler solve it. The magic word here is `constexpr`, which is available since C++11 and even more powerful with C++14.
  Some people might argue that macros should be used for this purpose, but in fact macros and `constexpr` are very different. Take a look at the following example:

```
#define macroAdd(x,y) ((x) + (y))
constexpr int exprAdd(int x, int y) { return x + y; };

int addOneAndTwo() {
  return 1 + 2;           // with no compiler optimizations enabled, the processor will add 1 and 2 each time
}

int useMacroAdd() {
  return macroAdd(1, 2);  // the compiler will replace "macroAdd(1, 2)" with "((1) + (2))" and the function be
}

int useExprAdd() {
  return exprAdd(1, 2);   // the compiler will replace "exprAdd(1, 2)" with "3"
}
```

  Furthermore, functions and variables that are defined using `constexpr` can be hidden in namespaces, which can prevent ambiguities in large projects.
  However, the disadvantage of `constexpr` functions is that the arguments must be known at compile-time.

```
int dynamicAdd(int x, int y, bool mode) {
  if (mode) {
    return macroAdd(x, y);  // this will work fine since this line is just converted to "return ((x) + (y));"
  } else {
    return exprAdd(x, y);   // this will fail, because the values of x and y are not known to the compiler
  }
```