AutoMATiC – Tutorial

Patryk Chaber

May 21, 2019

Contents

1 Introduction

2 Requirements

This software was tested using following setup:

- Microsoft Windows 10 Pro
- MATLAB 2019a
- MinGW64 Compiler (C++)

and for the programming of the exemplary microcontroller:

- STM32F746IGT6 microcontroller (part of the Core7XXI evaluation board with Open7XXI-C board – both produced by WaveShare)
- J-Link v9.3
- Keil uVision v5.17.0.0
- STM32CubeMX

Despite aforementioned configuration, this software was tested with MATLAB 2017b, thus should work with any other MATLAB version in between of those two. This software should work also on Linux based operating systems, but it may require minor changes in paths notation.

3 Installation

To build AutoMATiC system compile its sources using `mex` compiler in MATLAB with following steps:

1. download or clone the repository of AutoMATiC:
   `https://github.com/pjchaber/automatic-mpc.git`
2. make sure that MATLAB is callable from the shell by executing:
   ```
   matlab -batch "disp(version);"
   ```
   which should print output similar to the following:
   ```
   9.6.0.1072779 (R2019a)
   ```
   If the output differs, make sure that the following MATLAB directory:
   ```
   <MATLAB root directory>\bin\win64
   ```
   is listed in `PATH` variable of your system. For example:
   ```
   E:\Program Files\MATLAB\R2019a\bin\win64
   ```
3. register MATLAB for further use of its engine executing of the following instruction in shell:
   ```
   matlab -regserver
   ```
4. change current working directory to a directory with the source of the project
5. execute in MATLAB following instruction
   ```
   mex -v -client engine main.cpp XGetOpt.cpp -output automatic_transcompiler.
       exe;;
   ```
   If the compilation fails, make sure that you have set a C++ compiler by executing in MATLAB:
   ```
   mex -setup C++
   ```
   Also make sure, to use the same MATLAB version which was used in the shell at step 2 and 3.

4 Setup

All further examples focus on the task of controlling output voltage of emulated process. It is also done using voltage signals. All those signals range from 0 to 3.3V. From the implementation side, those signals are represented as values from -1 to 1, where the lowest value represents 0V and the highest one represents 3.3V. The process that is emulated is a matrix of single inertia transfer functions. Should I give an exact process definition or the code for one?

5 Single Algorithm in a Controller

Firstly lets create STM32CubeMX project which will be used as a base for further definition of a target platform's configuration (it is expected from the reader to know how to use STM32CubeMX and how to create projects for STM32F746IGT6 or similar, and how to, based on such a project, generate a Keil uVision 5 project). Details of this project are as follows:

- Target Platform: STM32F746IGTx
- Pinout and Configuration:
  - RCC (HSE as Crystal/Ceramic Resonator)
  - ADC3 (IN0 and IN1 enabled):
    - Clock Prescaler: PLCK divided by 4
    - Scan Conversion: Enabled
    - Continuous Conversion: Enabled
    - DMA Continuous Requests: Enabled
    - Number of Regular Conversions: 2
    - Rank 1:
      - Channel: Channel 0
      - Sampling Time: 144 Cycles
    - Rank 2:
      - Channel: Channel 1
      - Sampling Time: 144 Cycles
  - DAC (OUT1 and OUT2 enabled), for both outputs:
    - Output Buffer: Enable
    - Trigger: None
  - TIM2 (Clock Source: Internal Clock):
    - Prescaler: 1000-1
    - Counter Period: 10800
  - TIM5 (Clock Source: Internal Clock):
    - Prescaler: 0
    - Counter Period: 0xffffffff
  - USART1 (Asynchronous):
    - Data Direction: Transmit Only
  - DMA2
    - ADC3:
      - Stream: DMA2 Stream 0
      - Direction: Peripheral to Memory

            - Priority: Low
            - Mode: Circular
            - Peripheral Data Width: Word
            - Memory Data Width: Word
        * ADC3:
            - Stream: DMA2 Stream 0
            - Direction: Peripheral to Memory
            - Priority: Low
            - Mode: Circular
            - Peripheral Data Width: Word
            - Memory Data Width: Word
    - NVIC:
        * Time base of System tick Timer (Preemption Priority: 5)
        * ADC1, ADC2 and ADC3 global interrupts (Enable, Preemption Priority: 3)
        * TIM2 global interrupts (Enabled, Preemption Priority: 1)
        * USART1 global interrupts (Enabled)
        * DMA2 stream0 global interrupts (Preemption Priority: 4)

- Clock Configuration:
    - Input Frequency for HSE: 8Hz
    - PLL Source Mux: HSE
    - HCLK (MHz): 216 (confirm by pressing ENTER)

- Project Manager:
    - Toolchain / IDE: MDK-ARM V5
    - Minimum Heap Size: 0x8000
    - Minimum Stack Size: 0x40000

After generating code to a Keil uVision 5 project communication interface with the PC and interface to connect process of control has to be implemented. It is worth making sure, that all the code, that is added to the project after its generation is placed between comments `/* USER CODE BEGIN <name> */` and `/* USER CODE END <name> */`, which will further on be denoted as "<name>" block. It will further ease the process of regeneration of the microcontroller's configuration.

5.1 Communication with PC using USART

Firstly, in the `main.c` file, `string.h` library header has to be included in the "Includes" block. This will allow to send text messages to a PC in a simple manner. Next a simple wrapper for transmitting of a string should be defined in block "0":

```
void write_string(char * txt){
  while(HAL_UART_GetState(&huart1) == HAL_UART_STATE_BUSY_TX);
  if(HAL_UART_Transmit_IT(&huart1, (uint8_t*)txt, strlen(txt))!= HAL_OK)
    Error_Handler();
  while(HAL_UART_GetState(&huart1) == HAL_UART_STATE_BUSY_TX);
}
```

At this point, it is possible to send text messages to a PC, it is worth noting, that the new line character and carriage return is not included by default in the sent string.

5.2 Measuring Voltage Signal

Voltage signals after conversion to a digital values will be averaged over 100 consecutive samples. Therefore a table for those raw measurements have to be defined in block "PV" as:

```
uint32_t adc_val_raw[ADC_SIZE*2] = {0};
```

where `ADC_SIZE` is defined in block "PD" as:

```
#define ADC_SIZE 100
```

Due to having two signals, the size of table with raw measurements is `ADC_SIZE*2`. To increase the resolution of averaged measurements, those will be stored as `float` variables (block "PV"):

```
float adc_val_f[2] = {0,0};
```

Having those defined, it is important to start a ADC unit. It can be implemented by inserting following code into a block "2":

```
HAL_ADC_Start(&hadc3);
if(HAL_ADC_Start_DMA(&hadc3, (uint32_t*)adc_val_raw, ADC_SIZE*2) != HAL_OK)
    Error_Handler();
```

Thanks to a DMA mechanism, after obtaining 100 consecutive samples of each signal a specific callback function will be executed. It can be redefined to perform measurements' averaging as soon as they are available, by inserting following code in block "0":

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* AdcHandle)
{
  static int i=0;
  static uint32_t tmpval[2] = {0,0};
  for(i=0,tmpval[0]=0,tmpval[1]=0;i<ADC_SIZE; ++i){
    tmpval[0] += adc_val_raw[2*i];
    tmpval[1] += adc_val_raw[2*i+1];
  }
  adc_val_f[0] = ((float)tmpval[0]/ADC_SIZE-2047.5f)/2047.5f;
  adc_val_f[1] = ((float)tmpval[1]/ADC_SIZE-2047.5f)/2047.5f;
}
```

This will allow to have constant supply of averaged results from ADC without consuming much of a computational power of a microcontroller core (thanks to DMA).

5.3 Creating Voltage Signal

To generate voltage signal DAC are used. Those are already configured thanks to the generation of the code from STM32CubeMX, although, they have to be started before they can be used. To start them, in the block "2" there has to be invoked:

```
HAL_DAC_Start(&hdac, DAC_CHANNEL_1);
HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
```

A following calls allow to generate 0V on first output, and the 3.3V on the second one, since this is a 12 bit DAC:

```
HAL_DAC_SetValue(&hdac, DAC1_CHANNEL_1,DAC_ALIGN_12B_R,0);
HAL_DAC_SetValue(&hdac, DAC1_CHANNEL_2,DAC_ALIGN_12B_R,4095);
```

and thus, to use this with a variable that ranges from -1 to 1, a following function is defined (in block "WHILE"):

```
void __setControlValue(float* value){
  HAL_DAC_SetValue(&hdac, DAC1_CHANNEL_1,DAC_ALIGN_12B_R, (uint32_t)(value
      [0]*2047.5f+2047.5f));
  HAL_DAC_SetValue(&hdac, DAC1_CHANNEL_2,DAC_ALIGN_12B_R, (uint32_t)(value
      [1]*2047.5f+2047.5f));
}
```

where an array of two floats is an argument, and it contains values for consecutive outputs.

5.4 Inducing Interruptions with a Constant Frequency

To generate an interruption with a constant frequency, which will further be used as an indicator of the beginning of new discrete time, a timer TIM2 is used. It is already configured to generate

events with a constant period of 100ms, but it is not started yet. To do so, a following must be implemented in a block "2":

```
HAL_TIM_Base_Init(&htim2);
HAL_TIM_Base_Start_IT(&htim2);
```

This will cause an execution of a callback function which can be redefined:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if (htim->Instance == TIM2) {
    // interrupt each 100ms
    }
}
```

Although its implementation will be further extended.

5.5 Interface for Further Ease of Use

At this point it is worth to create additional functions, that will wrap already existing functionality, so that anyone who will be working with this project in terms of control algorithms, will have easy interface to work with. Firstly, to give access to measurements signals a following will be used (block "0"):

```
float* __measureOutput(){
  return adc_val_f;
}
```

and it will return a pointer to a two element array of measurement values. It is clearly visible, that it is analogous to the previously defined `void __setControlValue(float* value)` function. Those two functions have to have their declarations in the header file corresponding to the source file in which they are defined (block "EFP" in `main.h`). In this header, it could be convenient to declare also a function for communication with a PC `void write_string(char * txt)`.

Because the main function of the program (namely `int main(void)`) is defined in the AutoMATiC system software framework, the existing one have to be redefined using another name, e.g. `void low_lvl_main(void)`, which of course also has to be accessible from other source files. Therefore its declaration has to be given in a proper header file.

At the end it will be convenient to set up the Keil uVision 5 project in such a way, the transcompilation process is executed always right before the build process starts. For this, "Options for Target" window has to be opened, and in the "User" tab, in "Before Build/Rebuild" a "Run #1" has to be filled with an execution of the transcompiler. A following will be used:

```
<absolute path to AutoMATiC>/automatic_transcompiler.exe -I -p -d -i ../Src/
    main.mpc -o ../Src/main_mpc.c -l <absolute path to AutoMATiC>/Libs/MATLAB/ -
    L <absolute path to AutoMATiC>/Libs/C/
```

It is worth noting, that the transcompiler is executed while being in the directory with a project file (i.e. `MDK-ARM`). Consecutive arguments of the `automatic_transcompiler.exe` execution denotes:

- `-I` – usage of the interrupt based software framework variant,
- `-p` – usage of the profiler,
- `-d` – usage of the forced delay of the control signal application,
- `-i <path>` – relative path to the input file used for transcompilation,
- `-o <path>` – relative path to the output file storing the result of transcompilation,
- `-l <path>` – relative path to the directory with MATLAB libraries of the transcompiler,
- `-L <path>` – relative path to the directory with C libraries of the transcompiler.

To complete the setup of the project one has to add source files and headers of AutoMATiC system to a project for further compilation. All these files can be found in a directory `<absolute path to AutoMATiC>/Libs/C/`. Also a file named `main.mpc` has to be created and placed in the `../Src/` directory. This file will contain all the logic of the controller with as few low level implementation parts as possible.

5.6 Logic of the Controller

The file that contains logic of the controller, namely `main.mpc`, contains implementations which can be divided in a few parts:

- utilisation of the low level interface functions,
- invoke of the MATLAB script, used for automatic code generation of the MPC algorithm,
- implementation of controllers behaviour.

The execution of the MATLAB script is not necessary – content of this script could be as well placed right here. This notation nevertheless allows for clearer separation of the aforementioned parts of implementation of this file:

```
#include "stm32f7xx_hal.h"
#include <string.h>
#include "main.h"
#include "mat_lib.h"

long get_time(){ return HAL_GetTick(); }

extern void timer_loop(void);
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if (htim->Instance == TIM2) {
        timer_loop();
    }
}
void measurements(){
    #MPC_PROFILER_BEGIN 4
    new_output(&ad,__measureOutput());
    #MPC_PROFILER_END 4
}
void controls(){
    #MPC_PROFILER_BEGIN 3
    __setControlValue(last_control(&ad));
    #MPC_PROFILER_END 3
}
void hardware_setup(){
    #MPC_PROFILER_BEGIN 1
    low_lvl_main();
    #MPC_PROFILER_END 1
}

#MPC_BEGIN
% Here is the MATLAB code
algorithms_parameters_definitions
#MPC_END

ArchiveData ad;
CurrentControl cc;

void controller_setup(){
    #MPC_PROFILER_BEGIN 2
    init_archive_data(&ad, 200, 200, U_SIZE, Y_SIZE, 0, 0, 0.01);
    init_current_control(&cc,&ad);
    controller(NULL,NULL);
    #MPC_PROFILER_END 2
}

void idle(){
    #MPC_PROFILER_BEGIN 13
    const int k = 0;
  static char str[1000] = {0};
```

```
    sprintf(str, "x = [%f,%f,",ad.y[k][0], ad.y[k][1]);     write_string(str);
    sprintf(str,       "%f,%f,",ad.z[0], ad.z[1]);          write_string(str);
    sprintf(str,       "%f,%f,",ad.u[k-1][0],ad.u[k-1][1]); write_string(str);
    write_string("];\n\r");
    #MPC_PROFILER_END 13
}

void loop(){
    #MPC_PROFILER_BEGIN 10
    static int i = 0;
  if(i<  50){ ad.z[0] = -0.1; ad.z[1] =  0.2; }
  else      { ad.z[0] =  0.1; ad.z[1] = -0.2; }
    if(++i > 100) i = 0;

    #MPC_PROFILER_BEGIN 50
  controller(&ad,&cc);
  #MPC_PROFILER_END 50

    push_current_controls_to_archive_data(&cc,&ad);
    #MPC_PROFILER_END 10
}

void timeout(){
    while(1);
}
```

On the listing, there is a reference to a function called `controller`, which declaration is as follows:

```
void controller(ArchiveData * ad, CurrentControl * c);
```

This function will be generated as soon as the MATLAB script is finished.

The AutoMATiC system software framework assumes that there is an implementation of functions:

- `void write_string(char * str);` – function used by the profiler to write out the results of its measurements,
- `long get_time(void);` – function used by the profiler to measure time of execution,
- `void hardware_setup(void);` – function used to setup hardware configuration of the controller,
- `void controller_setup(void);` – function used to setup software configuration of the controller,
- `void measurements(void);` – function used to obtain measurements for the current controller iteration,
- `void loop(void);` – function used to define the behaviour of the controller – here control algorithms are executed,
- `void controls(void);` – function used to determine what to do with results obtained from consecutive control algorithms,
- `void idle(void);` – function used for other, lower prioritized procedures,
- `void timeout(void);` – function executed when the iteration lasts longer than the sampling period.

These functions are used in the framework, in the predefined order. If some of these are not implemented, a default (often empty) implementation is assumed.

To implement interrupt based variant of the software framework, a function

```
extern void timer_loop(void);
```
has to be declared, and called each time a new discrete time instant starts. In these examples it occurs each time, the timer TIM2 generates an interrupt, and thus causes execution of the following function:
```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);
```
Each new measurement has to be stored in the `ArchiveData` structure, using
```
void new_output(ArchiveDataPtr ad, float* y);
```
function. Also the final values of new control signals have to be stored in the same structure using
```
void push_current_controls_to_archive_data(CurrentControlPtr c,ArchiveDataPtr
    ad);
```
where `CurrentControl` structure is filled by each control algorithm.

Lastly, there are a few functions in the `void controller_setup()` function, that are necessary to use those two structures. A function
```
void init_archive_data(ArchiveDataPtr ad,
  long number_of_u, long number_of_y,
  long size_of_u, long size_of_y,
  float default_u, float default_y,
  float current_yzad);
```
is used to initialize the `ArchiveData` instance. First argument is a pointer to this structure instance, next there are two parameters used to define number of control and output signals (in this order) of the process of control. Further there are two values that determine, how many previous values of control and output signals have to be stored in the memory of the controller. At the end, there are default values of control, output and set-point signal, used for initialization of the microcontroller memory – those can be changed as soon as the initialization function is finished.

Next function is
```
void init_current_control(CurrentControlPtr cc, ArchiveDataPtr ad);
```
which is used to allocate memory for an instance of `CurrentControl` based on the data stored in the alreaty setup `ArchiveData` instance.

At the end of the software setup, there is an execution of the controller function, with both parameters set to `NULL`, which is used to initialize some values of the controller that are constant through its whole existence – this will be clear as soon as the generated code of the algorithm appears.

5.7 Dynamic Matrix Control – Analytic version

5.8 Dynamic Matrix Control – Numeric version

5.9 Generalized Predictive Control – Analytic version

5.10 Generalized Predictive Control – Numeric version

6 Multiple Algorithms in a Controller

6.1 Analytic and Numeric versions of DMC

7 Extending of the Library

7.1 Proportional-Integral-Derivative Controller