

# Falcon BMS to Arduino Interface Tool (BMSAIT)



Author	Robin "Hummer" Bruns
Document version	1.4.2
BMSAIT version	1.4.4
BMS version	4.37
Date	21.04.2025

## Content

1.	Introduction.....	5
1.1.	Fundamental.....	5
1.2.	Quickstart .....	6
2.	Basic logics.....	7
2.1.	Data exchange between PC and Arduino board .....	7
2.1.1.	Hardware Connection.....	7
2.1.2.	Software Connection .....	7
2.1.3.	PUSH principle .....	8
2.1.4.	PULL principle .....	10
2.2.	Synchronization .....	13
2.2.1.	Engine calibration .....	13
2.2.2.	Bring engines to zero position.....	13
2.2.3.	Synchronization of switch positions.....	13
3.	Windows Application .....	15
3.1.	Getting Started .....	15
3.2.	BMSAIT Windows App.....	16
3.2.1.	The main form .....	16
3.2.2.	Basic Settings.....	21
3.2.3.	Set up devices.....	23
3.2.4.	Variable Selection Form .....	24
3.2.5.	Form Input Commands.....	26
3.3.	(optional) Activate virtual joysticks .....	28
3.4.	(work in progress) DCS Integration .....	29
3.5.	GaugeTable.....	30
4.	The Arduino Program .....	32
4.1.	Description of the preparations .....	32
4.1.1.	Foreword .....	32
4.1.2.	Programming an Arduino with the BMSAIT Sketch.....	32
4.1.3.	Download the program code .....	32
4.1.4.	Installation of a development environment .....	32
4.1.5.	Choosing the Arduino Board .....	34
4.1.6.	COM Port Selection .....	34
4.1.7.	Checking the Arduino software .....	34
4.1.8.	Uploading the Arduino software .....	35

4.2.	Description of the Arduino Sketch .....	35
4.2.1.	The module BMSAIT_Vanilla .....	35
4.2.2.	The UserConfig module .....	36
4.2.3.	The Switches Module .....	40
4.2.4.	The ButtonMatrix module .....	43
4.2.5.	The Encoder Module .....	45
4.2.6.	The Analog Axis Module .....	44
4.2.7.	The LED module.....	47
4.2.8.	The LED matrix module .....	49
4.2.9.	The LCD module.....	50
4.2.10.	The SSegMAX7219 module .....	51
4.2.11.	The SSegTM1367 module .....	51
4.2.12.	The Servo Module .....	52
4.2.13.	The ServoPWMSHIELD module.....	53
4.2.14.	The Stepper module .....	55
4.2.15.	The StepperX27 module .....	56
4.2.16.	The Air Core module.....	58
4.2.17.	The StepperVID module .....	57
4.2.18.	The MotorPoti module .....	59
4.2.19.	The OLED module .....	60
4.2.20.	The Speedbrake Indicator module .....	62
4.2.21.	The FFI module .....	64
4.2.22.	The DED/PFL module.....	65
4.2.23.	The Backlighting module .....	66
5.	Grounds .....	<b>Fehler! Textmarke nicht definiert.</b>
5.1.	Sources / References .....	67
5.2.	Data Field Descriptions.....	68
5.2.1.	Data variables (BMSAIT-Variablen.csv, Windows app) .....	68
5.2.2.	Variable mapping (Windows app) .....	69
5.2.3.	Commands (Windows App).....	69
5.2.4.	Global Variables (Arduino) .....	70
5.2.5.	Internal Commands .....	72
5.2.6.	Font DED/PFL.....	73

Changelog		
1.1	11.04.2020	Various adjustments
1.2	17.12.2020	Various adjustments
1.3	31.01.2021	New: DCS integration New: OLED and FFI modules Editorial adjustments
1.4	17.01.2023	New: Data Monitor New: Gauge Table Additions to the LED/LED Matrix modules
1.4.1	24.07.2024	New: Air Core module
1.4.2	21.04.2025	New: Module Magnetic Switch Reworked DCS integration

## 1. Introduction

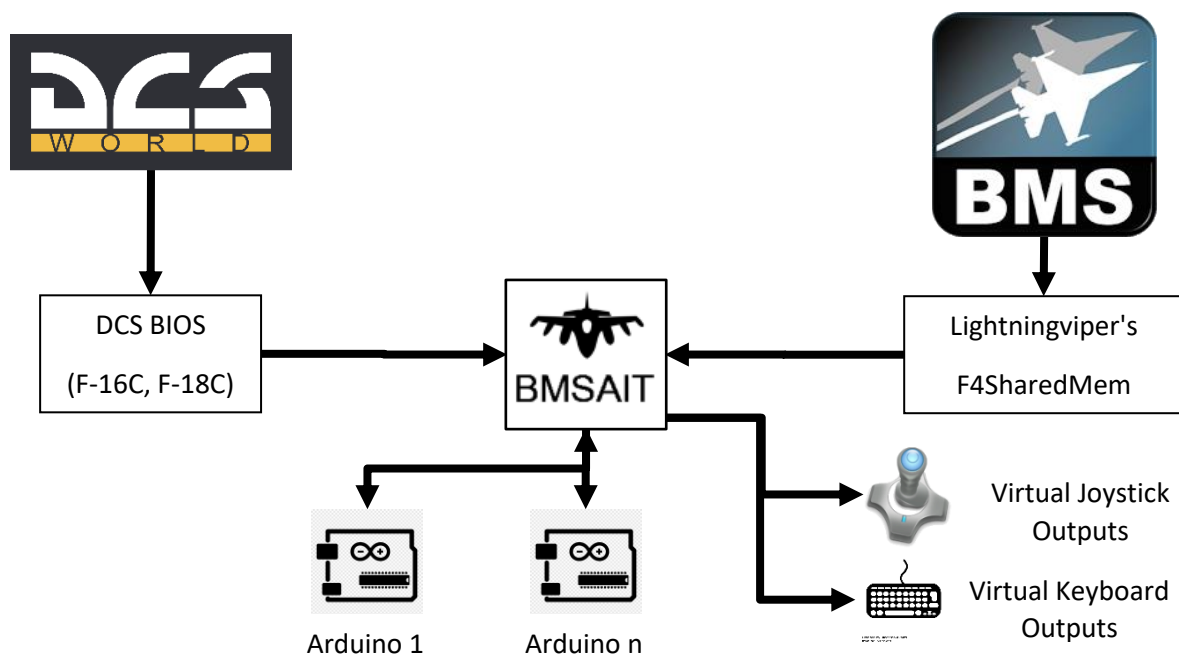
### 1.1.Fundamentals

The Falcon BMS to Arduino Interface Tool (BMSAIT) is a tool for extracting flight information from Falcon BMS and making it available for controlling devices during cockpit construction, as well as for processing and sending control signals to flight simulation. The tool is divided into two components:

1. The windows program (windows app)
  - a. extracts data from Falcon BMS's shared memory and outputs this data via a serial port.
  - b. Receives control information from an Arduino and converts it into keyboard signals or joystick commands.
2. The Arduino sketches configure an Arduino board to
  - a. Receive data via the serial port and output it via selected devices (LEDs, LCDs, servos, others).
  - b. Read out different input devices and send control signals back to the Windows program.

Many tools have already been provided in the past for extracting data from BMS SharedMem (just to name a few: FAST, BMSCockpit, DEDuino, F4toSerial).

In many areas, BMSAIT cannot and should not compete with these programs. BMSAIT was developed out of necessity that many of the tools mentioned are either no longer developed (and thus not all the data available in the SharedMem of Falcon BMS with the latest updates are available) or have a limited range of functions and thus cannot support all cockpit projects. In the current stage of development, BMSAIT can therefore be seen as a supplement to use cases that are not covered by the common programs.



The use of BMSAIT was initially intended as a pure output program; the generation of control signals via input devices was only a marginal function. However, with the further development of input functions, there is great potential for an Arduino with BMSAIT software to theoretically be able to map all the functions of a panel when building a cockpit. Cabling cockpit panels to central controllers can lead to complex and possibly error-prone structures (I know what I'm talking about here!) BMSAIT allows one or more panels to be connected to just one Arduino, located directly behind the panel, which covers all inputs and outputs of the panel. To the outside, the panel only needs a single USB cable (and possibly an external power supply). This helps to make the cockpit construction clear.

Basically, it is intended that the BMSAIT Windows app will be used in combination with the Arduino sketches provided. However, it is theoretically possible to use both tools independently of each other in conjunction with other software products.

If you want to use BMSAIT, you will have to deal with hardware issues and a little bit with programming Arduinos using C++. With this documentation, I hope to provide all the necessary explanations to enable you to set up the software even without programming knowledge. If in doubt, please feel free to contact me - within the framework of free time capacities, I will be happy to support you with the setup or your ideas for extensions.

BMSAIT can theoretically be used with all common Arduino boards. However, I have noticed problems in communication with the Leonardo/Pro Micro, so I recommend using the Uno, Micro, Nano or Mega. Support for the Due has been available since version 1.2. However, the peculiarities of this type can lead to problems.

For some applications, it is advantageous if the microcontroller has more power and more memory (e.g. to control the DED). For this purpose, it is also possible to use BMSAIT on controllers such as the ESP32.

## 1.2.Quickstart

To get started, read chapters 3.1 (Getting Started with Windows Application) and Chapter 4.1 (Preparing for Arduino) to set the stage for using BMSAIT.

For the first attempts with BMSAIT, I recommend trying the available [examples](#) . The example programs include a ready-made sketch for the Arduino (\*.ino and the associated .h files), a configuration file (\*.ini) for the Windows app, and separate documentation with a detailed description of the required cabling.

If the functionality of the software is proven on your computer and you have understood the basic features of the software, it will be easier for you to implement your own projects.

## 2. Basics

### 2.1. Data exchange between PC and Arduino board

#### 2.1.1. Hardware connection

Data exchange between Windows and Arduino takes place via a serial data connection. This is usually achieved via the Arduino's USB connection to the Windows computer.

The Arduino board is assigned a COM port. As long as the board is always connected to the same USB port, the board will be assigned the same COM port number. If you connect the Arduino board to a different USB port, it must be noted that the board will be assigned a different COM port.

The Arduino board is powered by the USB port via the PC. If the Arduino is used to control devices with high power requirements (e.g. servo motors), an additional power supply should also be used to prevent damage to the Arduino board.

Note: If you are using additional power sources for a device connected to the Arduino board, please make sure that the ground connections of the Arduino and the power source are connected!

#### 2.1.2. Software connection

In addition to the hardware connection, software settings are also a prerequisite for communication between Windows and an Arduino board. The first step here is to choose the BAUD rate (the transfer speed). The BAUD rate is specified for the Arduino in the program running on it. If you want to change the BAUD rate, you have to adjust the Arduino Sketch and upload it again (see 4.2.2).

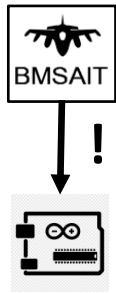
In the Windows app, the BAUD rate can be set before establishing a connection in the device settings (see 3.2.4). Communication is only possible if both BAUD rates match. It is possible to set an individual connection speed for each Arduino.

BMSAIT supports two different forms of communication for exchanging information with appropriately programmed Arduino boards:

- The PUSH mode
- The PULL mode

Since version 1.3, BMSAIT is not limited to one of the methods, but it is possible to define the processing logic for each Arduino individually.

### 2.1.3. PUSH mode



When using the PUSH mode, the Windows app is in charge of the data exchange. The relevant information of the SharedMem from Falcon BMS is selected in the Windows app. The app has an editor for this purpose, which can be used to select and configure the desired data (see 3.2.5).

The Arduino board only accepts data and displays it. The advantage here is that the Arduino board is relieved of some computing power and testing of new functions is easier.

#### 2.1.3.1. Flow logic of the Windows software during PUSH mode

##### Start of processing:

By clicking on the "Start connection" button, the processing of the following steps begins. If "autostart" has been activated in the basic settings, processing will start immediately when the Windows app is started.

##### Initialization:

The initialization takes place once when the connection is established

- Quick access to selected SharedMem variables is established
- Opening the COM ports to the Arduinos

##### Loop:

The loop runs endlessly until the user terminates the connection. The frequency of the loop pass can be determined by the value *polltime*. It can be regulated in the basic settings (see 4.2.2).

- Load the latest image of the SharedMem from the Falcon BMS
- Display the data on the data monitor
- For each data variable configured in the Windows app, the following procedure is followed:
  - Based on the data type and ID of the data variable, the desired value is read from the image of the shareMem
  - The read value is converted to ASCII code
  - A data package for transmission is prepared (addition of control information to the data variable)
  - The data packet sent via the serial port
- Listen for an Arduino response on a COM port.
  - Messages are output in the text console of the main form
  - Instructions for keyboard commands are processed and key signals are triggered

##### End of processing:

Clicking on the "End connection" button stops processing.

Closing the program BMSAIT also obviously stops processing.

#### 2.1.3.2. Arduino Software Flow Logic during PUSH mode

##### Start of processing:

Processing begins as soon as the Arduino receives a power supply and a valid program is loaded.

##### Initialization (happens every time a new connection is established):

- Pre-populating common variables



- Start software for controlling the peripherals (integrating libraries and assigning variables)
- Open COM port to listen for signals from the Windows app

**Loop:**

The loop runs endlessly

- Check the position of connected switches/analog axes.  
If changed: Send commands to the PC
- Listen for system commands from the PC  
If so, these commands will be processed
- Listen for simulation data from the PC  
If data is available: Read and cache the new data
- Update data on connected devices

**End of processing:**

Processing can only be terminated by disconnecting the Arduino from the power supply or by overwriting the existing program with another program.

### 2.1.3.3. Record format of a transfer from PC to Arduino using the PUSH principle

<START-Byte><INFO-Byte><Formatstring1><Data><Formatstring2>

**START-Byte:**

The byte 255 signals the beginning of a transmission (see 4.2.2)

**INFO-Byte:**

The Infobyte tells the Arduino what kind of message it is:

Handshake:

If a defined byte is transferred here (see 4.2.2 "HANDSHAKE"), this is a sign for the Arduino that the Windows app is currently looking for connected Arduino boards. The Arduino is equipped with a defined response signal (ID of the board, see 4.2.2.2) to indicate to the Windows app that the message has been received and the board is ready.

Control PULL:

See 2.1.4.4.

Data transmission:

In the case of a data transfer, the INFO byte is used to indicate the position number in which the value of the current transmission is to be temporarily stored in the prepared data container of the Arduino.

**Formatstring1:**

In the Windows app, formatting information can be specified for the variable assignment. The format string1 contains the characters that should come before the actual data value. By default, the BMSAIT Arduino sketch expects the character '<' here (see 4.2.2 "VAR\_BEGIN").

**Data:**

This is the information read out of the SharedMem. This can be a single character (e.g. a true/false value) or a string with many characters (e.g. one line of the DED). The maximum length for a dataset

can be set in the Arduino sketch. The value should range from 1 (excluding true/false LED processing) to 25 (length of a line in the DED or PFD).

#### Formatstring2:

In the Windows app, formatting information can be included in the variable assignment. The format string2 contains the characters that should be placed after the actual data value. The included Arduino sketch expects the character '>' here. (see 4.2.2 "VAR\_ENDE").

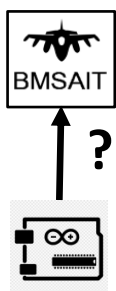
#### 2.1.3.4. Dataset format of a transfer from Arduino to PC using the PUSH principle

A transfer consists of specifying the transfer type and the actual data set:

<Transfer Type>< Record>

Transmission Type	Content Dataset
t	The transferred record should be displayed in the BMSAIT console
k	The current state of a switch is transmitted (1- active, 0- not active)
a	<p>The transmitted data set contains the position of an analog axis: &lt;Command ID&gt;,&lt;Axis Position&gt;</p> <p>The command ID is a three-digit number that refers to an entry in the input command management of the Windows app. When a command with this ID is assigned to the Arduino in the Windows app and connected to an analog axis of a vjoy, the vJoy analog axis is moved to the position transmitted by the Arduino (0..1024).</p>
g	The Arduino reports that the input buffer is empty and therefore new data can be received.
s	The Arduino sends an internal command to the BMSAIT app

#### 2.1.4. PULL mode



In the PULL mode, the Arduino software is in charge of data transmissions. The Windows software listens for requests from the Arduino and only sends data when the Arduino explicitly requests it. The advantage here is that, in contrast to the PUSH mode, you only have to set up the configuration of the desired data variable once (namely only in the Arduino). PULL also facilitates the synchronicity of data exchange, so that time-critical data reaches the Arduino more quickly. All required data variables must be entered in the Arduino sketch (BMSAIT-UserKonfig.h) – as with the PUSH.

##### 2.1.4.1. Windows software flow logic during PULL mode

PULL processing can be selected in the Windows app in the settings of an Arduino device.

#### Start of processing:

Clicking on the "Launch!" button in the Windows app will start processing on the Arduinos that have been set to PULL processing:

**Initialization:**

The initialization takes place once when the connection is established.

- The Windows app sends a code signal to the Arduino to activate the PULL logic.
- A connection is established to the sharedMem of BMS or DCS

**Loop:**

- Read current data from the SharedMem and cache it internally
- Output the SharedMem data to the data monitor
- Listen for requests or commands from the Arduino
  - Output messages in the text console of the Windows App
  - Processing data requests:
    - Reading the corresponding value according to the data request from the SharedMem
    - Converting the read value to ASCII code
    - Preparation of the data package for transmission (addition of control information to the data variables)
    - Sending the data packet over the serial port
  - Triggering keypad/joystick signals
  - Moving analog axes

**End of processing:**

Clicking on the "End connection" button stops the processing. To do this, a signal is transmitted to the Arduino, which causes it to stop sending PULL requests.

#### 2.1.4.2. [Arduino software flow logic during PULL](#)

**Start of processing:**

Processing begins as soon as the Arduino receives a power supply and a valid program is loaded.

**Initialization:**

- Presetting variables
- Start software for controlling the peripherals
- Open COM port

**Loop:**

The loop is run by the Arduino to the end of processing.

- Listen for commands from the PC
  - If so, the commands are processed
  - When the command to activate the PULL principle is received, the following PULL processing is processed
  - If the command to disable the PULL principle is received, the following PULL processing will no longer be processed
- PULL processing (only if enabled)
  - Iterate through all data variables entered in the data container
  - Preparation of a command for the data request for the data variable
  - Sending the data request
- Listen for serial data from the PC
  - If data is available: read and cache data
- Output cached data from the connected periphery

- Check the position of connected switches: Commands are sent to the PC when a change is made

**End of processing:**

Processing can only be terminated by unplugging the Arduino.

#### 2.1.4.3. Dataset format of a transfer from Arduino to PC using the PULL principle

In addition to the transmission format according to PUSH (see 2.1.3.4) the transmission types "d" and "u" are used here to send data requests to the PC:

d	<p>The transferred record contains a data request in the format: &lt;Data Position&gt;.&lt;Variable Type&gt;.&lt;VariableID&gt;</p> <p><b>Data Location</b> This is the position in the data container of the Arduino sketch</p> <p><b>Variable</b> Here, the variable type of the data variable is displayed with one byte, e.g. {f} for a float number. The value is read from the data container (datafield.format). If an incorrect format is specified in the data container, the request will not be able to be processed by the Windows app. See also the description of the <i>data variable</i> in the appendix.</p> <p><b>VariableID</b> Here, the ID of the data variable is taken from the data container (Datenfeld.id). For further processing, it is important that it is a text with exactly four characters. When filling the data container, it is therefore important to make sure that leading zeros are included!</p>
u	<p>This command asks the Windows app to transfer the current data of the DED ("uDED") or PFL ("uPFL").</p> <p>The WinApp will then process the five lines of the DED/PFL and send them one after the other to the requesting Arduino.</p>

#### 2.1.4.4. Dataset format of a transfer from PC to Arduino using the PULL principle

The format according to chapter 2.1.3.3. with the addition that two info bytes are defined that control the Arduino's PULL processing:

**Start PULL processing:**

If the value 0xAA (170) is transmitted as an info byte, this is a sign for the Arduino that the PULL processing should be started (see 5.2.4 "STARTPULL").

**Stop PULL processing:**

If the value 0xB4 (180) is transmitted as an info byte, this is a sign to the Arduino that the PULL processing should be stopped (see 5.2.4 "ENDPULL").

## 2.2. Synchronization of hardware devices

### 2.2.1. Motor calibration

If you want to use stepper motors in your cockpit, the software is unable to store the current position of the motors. When the Arduino is restarted, the software therefore does not know in which position a pointer of a stepper motor is currently located.

For this reason, stepper motors must be calibrated before use in the simulation. This will bring the motors/gauges into a position in which the software and hardware positions match.

An engine calibration involves a motor being driven to the zero position (full counterclockwise deflection), then performing the full range of motion clockwise and then returning to the zero position. This is to ensure that the motors/gauges always end in the respective zero position. This only works with small torque motors (i.e. x27.168) that can be blocked by a physical stop without taking damage.

Calibration can be done in three ways:

On the main form of the BMSAIT Windows app, the "calibrate motors" button can be pressed. This sends a signal to all defined Arduinos and instructs them to calibrate the connected motors.

The selection menu (right click) of the configured devices can be used to instruct a specific calibration of a specific Arduino.

A button on an Arduino configured with BMSAIT can be set up with an internal command (255). This will send a request to the Windows app. The Windows app will then instruct all Arduinos to perform a motor calibration.

### 2.2.2. Bring motors to zero position

A complete motor calibration is not appropriate in some cases because it takes too long. Therefore, there is the possibility of performing a "small" calibration, in which the motors are only driven into the assumed zero position.

This can be achieved in the following ways:

1. Click on the "calibrate motors" button on the main form of the BMSAIT Windows app,
2. Automated calibration is also provided. This is triggered when exiting the 3D world of BMS or when calling up the 2D world of DCS,
3. A button connected to an Arduino can be used to send the request for a "small" synchronization to the Windows app via an internal command (254).

### 2.2.3. Synchronization of switch positions

When starting the simulation, the switch positions in the simulation may not match those in the home cockpit. It might be necessary to move a switch physically once to ensure that the hardware and software position match.

To achieve a quick synchronization, it is possible to instruct the Arduinos to pass the position of all switches once. This can be used to transfer the position of the physical switches to the simulation.

There are three ways to do this (which, for obvious reasons, only make sense if you are in the 3D world of the flight simulation):

1. Click on the "calibrate motors" button on the main form of the BMSAIT Windows app,
2. The synchronization is automatically triggered when entering the 3D world of BMS,
3. A button connected to an Arduino can be used to send the request for synchronization to the Windows app via an internal command (253).

### 3. Windows Application

The Windows application was written using Windows Visual Studio 2019 as a Windows app in C# for x64 processors using the .net components version 4.7.2.

The following libraries are used for operation:

LightningViper's "F4SharedMem.dll" to access the Falcon BMS SharedMem

"WindowsInputLib.dll" (Input Simulator Plus) by Michael Noonan, Theodoros Chatzigiannakis

"vJoyInterface.dll" and "vJoyInterfaceWrap.dll" for generating joystick commands

On request, I will be happy to provide the project files with my source code.

#### 3.1. Getting Started

To start the program, the following files are required, which must be placed together in a folder of the Windows system:

- BMSAIT.exe
- vJoyInterface.dll
- BMSAITVariablen.csv
- BMSAIT-GaugeTable.csv
- Subfolder "en" with the file BMSAIT.resources.dll for English language settings

The following files have been embedded in the BMSAIT.exe since version 1.4 and are no longer needed to be placed in the BMSAIT folder separately:

- F4SharedMem.dll
- WindowsInput.dll
- WindowsInputLib.dll
- vJoyInterfaceWrap.dll
- Subfolder "de" with the file BMSAIT.resources.dll for German language settings

When you start the program for the first time, you will get error messages, because the basic settings must be set first.

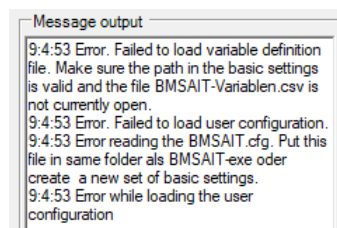


Figure 1: Error message when basic configuration is missing

The update of the basic settings is described in chapter 3.2.2 described.

## 3.2.BMSAIT Windows App

### 3.2.1. The main form

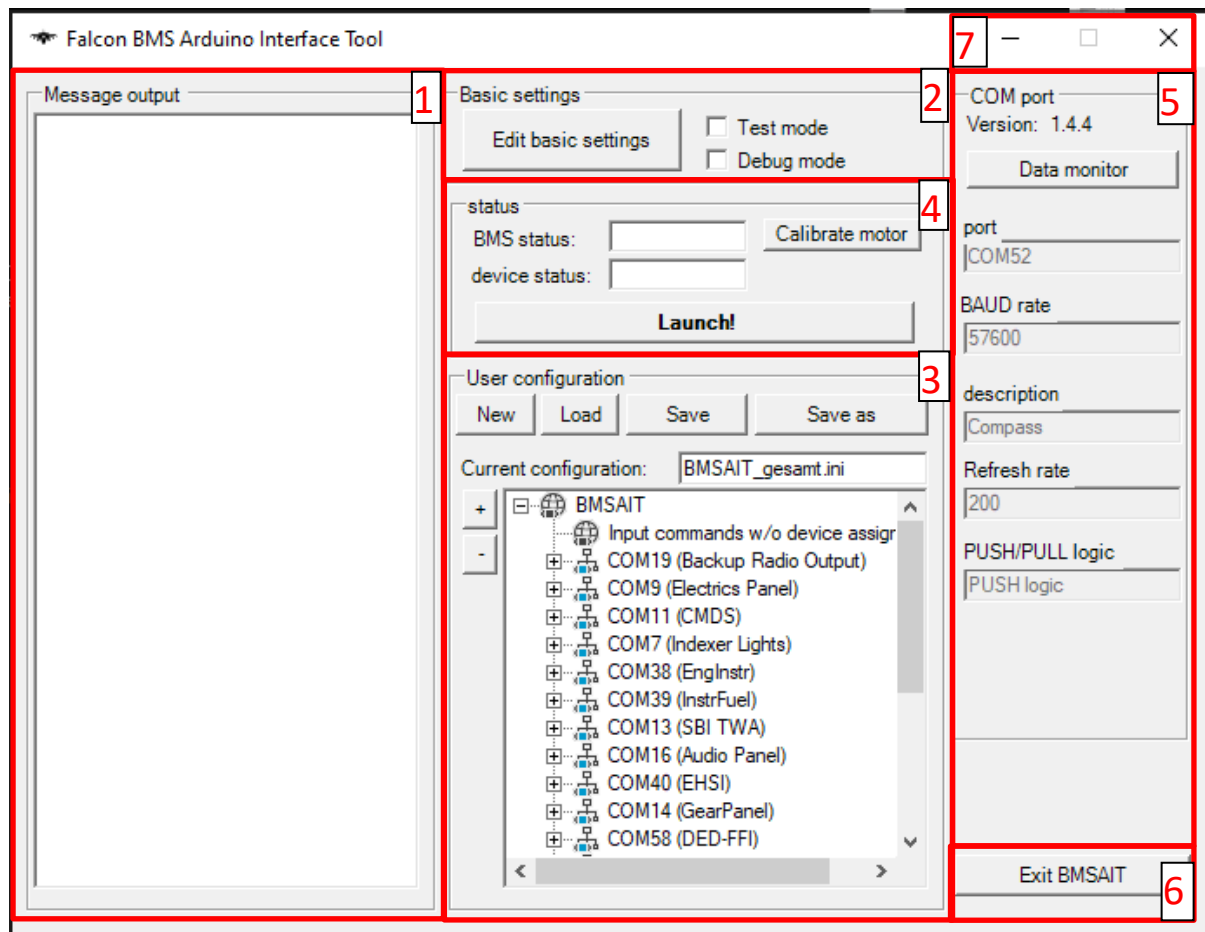
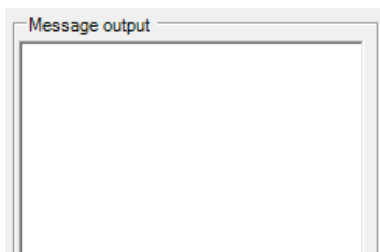


Figure 2: BMSAIT Main Form

When the program starts, the main form is invoked. The main form is used to provide information about the current settings and allows you to control the application.

The main form is divided into sections.

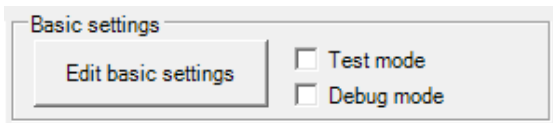
#### Main Form Section 1 - Console Output



This field displays status averages and error messages.



## Main Form Area 2 – Basic Configuration

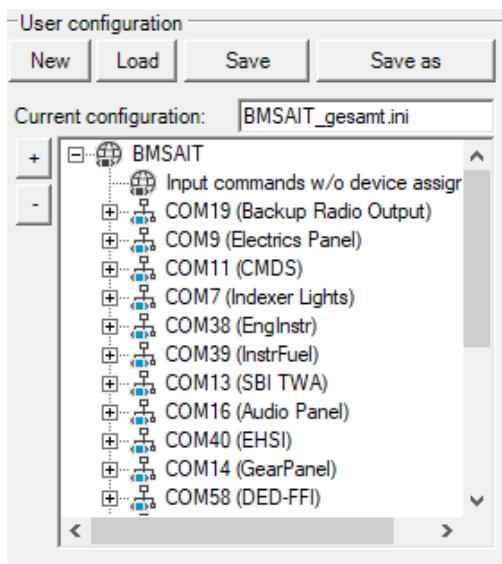


The "Edit basic settings" button opens a form in which the basic settings of the program can be adjusted.

The test mode flag offers the possibility to test a data connection of the Windows app with connected Arduinos without having to start Falcon BMS. If this flag is set and processing is started, no connection is established with the SharedMem of the Falcon BMS. Instead, it uses the test data entered in the configuration when defining a data variable. In test mode, the input commands reported by the Arduinos are also triggered, even if the Falcon BMS and DCS are not started.

When debug mode is enabled, detailed Arduino status and error messages are displayed in the console for analysis after processing begins.

## Main Form Section 3 – User Configuration



In this area, tools for creating a configuration of devices, data variables of the sharedMem and input commands are displayed.

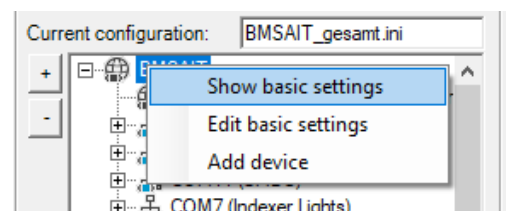
Three levels will appear in the window. Level 1 is the TOP node. Below the TOP node, devices (COM ports representing an Arduino) are displayed. At the lowest level, the variables and input commands assigned to them are displayed.

The "Input commands without device assignment" node displays input commands that have not been specifically assigned to an Arduino. Since version 1.3, input commands are directly assigned to an Arduino. If input commands are assigned to this node, you have tried to

load an outdated configuration. For new configurations, this node is irrelevant.

A left mouse click on an entry in the user configuration window selects it. Section 5 of the main form displays detailed information about the selected object.

By right-clicking on the **TOP node**, the following options are available:



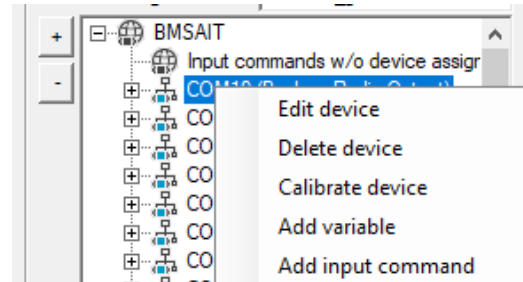
### View/Edit Basic Settings

A form will open in which the basic settings of the program can be adjusted.

### Add device

A window will open to add an Arduino.

By right-clicking on a **device** , the following options are available:



### Edit device

A window will open in which the current settings of the selected device can be displayed and adjusted.

### Delete device

The selected device including all variable assignments is deleted.

### Calibrate device

The motors are recalibrated on the selected device.

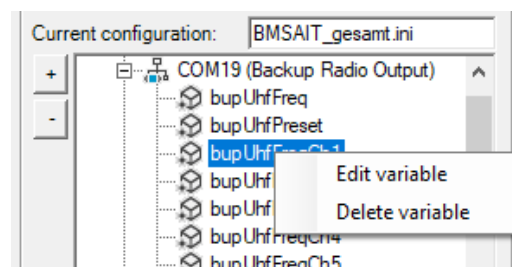
### Add variable

A window will open to add a variable

### Add input command

A window for adding input commands will open

By right-clicking on a **variable**, the following options are available (only with the push principle):



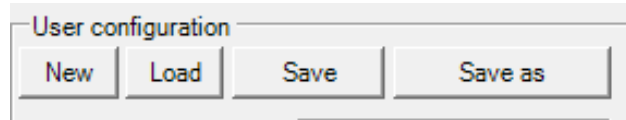
### Edit Variable

A window opens in which the current settings of the selected variable can be displayed and adjusted.

### Delete Variable

The assignment of the variable to the device is deleted.

In section 3 there are the following buttons:



### New

The current user configuration is deleted and replaced by an empty configuration.

### Load

A file selection dialog is opened, which can be used to select a user configuration. If a valid configuration is selected, the corresponding data in this area is updated.

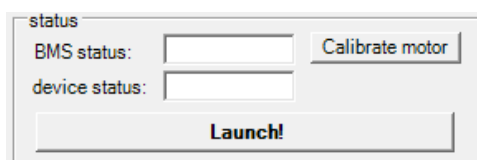
### Save

The current user configuration is saved at the default storage location. (The default storage location is stored in the basic settings)

### Save As

The current user configuration is saved in a new file. A file selection dialog is opened, which can be used to select the desired storage location and the file name.

## Main Form Area 4 – Processing Control



In this area, the actual processing logic of the program is started.

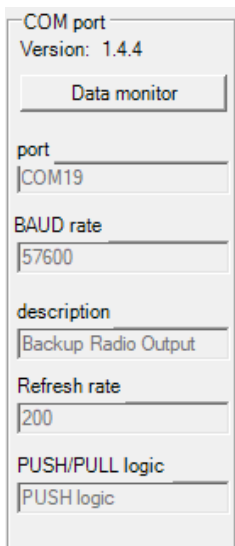
The "BMS status" field shows the connection status with Falcon BMS or DCS. A distinction is made between *no connection*, *connection (2D world)* and *connection (3D world)*. BMSAIT also detects whether the Alternate Launcher of BMS is currently running.

The "device status" field indicates whether a COM connection can be established with the Arduino(s) after initiation of the processing. The indicator is green if there is a connection to all defined Arduinos. If no connection is possible, the indicator is red. If several Arduinos have been defined and it has not been possible to establish a connection to all of them, the indicator is yellow.

The button "Calibrating motors" establishes a temporary connection to all Arduinos and sends the command to calibrate all connected devices. This is particularly relevant when using stepper motors to bring the position of these motors to the zero position (see chapter 2.2).

The "Launch!" button triggers the main loop of the program. The main loop continuously reads the current data from the SharedMem of the Falcon BMS / DCS BIOS and manages the data transfer to the Arduino(s) and the initiation of signals via a joystick/keyboard controller.

## Main Form Section 5 – Detailed Information

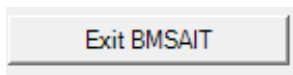


This section displays detailed information. The display depends on whether the TOP node, a device, variable or input command has been selected in the configuration window (area3).

At the top, the current software version of BMSAIT is displayed.

This can be started by clicking on the "Data Monitor" button (see 3.2.2).

## Main Form Area 6 – Exit Program



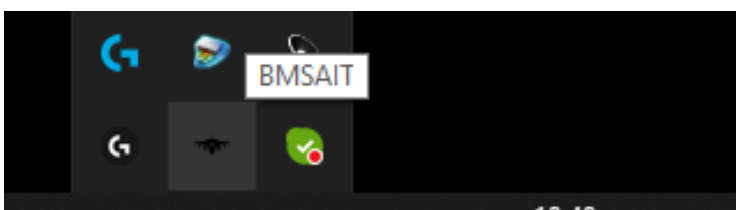
This is where the program ends.

## Main Form Area 7 – Window Management

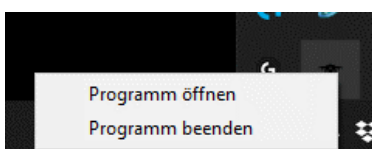


Here the program can be ended by clicking on the cross.

Clicking on the left line minimizes the display of the program. Ongoing processing continues.



The BMSAIT program will then be displayed as an icon in the taskbar.



To open the program as a window again, right-click on the icon to select the command "Program öffnen".

### 3.2.2. Data Monitor

The Data Monitor window displays the following data fields:

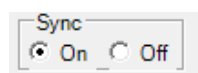
Tab	Field	Value
A/C Data	X	2372882
	Y	1566340
	Z	-10027,12
	xDot	-565,6316
	yDot	239,0508
	zDot	0,1852868
	AOA	3,53669
	Beta	0,005939909
	Gamma	-0,000297061
	EngineData	Pitch
Roll		0,0004696
Yaw		2,763686
Altitude	Indicated	-10552,81
	AltDigits	1055
	CabinAlt	7869,713
	RadarAlt	-999999,9
	Alt Calibr.	1005
Speeds	Mach	0,5980324
	KIAS	327,7135
	TAS	626,8665
	GS	0,9985957
MagDevSystem	MagDevSystem	0
	MagDevReal	0
	WindOffset	-0,0215873
Trim	TrimPitch	0
	TrimRoll	0
	TrimYaw	0
Controls	<input type="checkbox"/> On Ground	
	<input type="checkbox"/> Autopilot on	

At the bottom, there are version fields: Flightdata1 Version: 118, Flightdata2 Version: 21, a Sync section with radio buttons for On (selected) and Off, and a Close button.

Figure 3: Data Monitor

The data monitor can be called up via the main window.

The data monitor includes almost all data fields available in the SharedMem. For reasons of clarity, these have been divided into different tabs.



The data monitor includes two functions that are controlled by the synchronization mode.

#### Sync = on

The data monitor is used to display the current data read from the shareMem of BMS/DCS and can be used for troubleshooting if displays do not behave as desired. Data fields cannot be changed in synchronized mode.

#### Sync=off

The data monitor serves as a console for entering your own data values. The data fields can be edited. Changes are transmitted to the Arduinos and can be used to check functions (LED, motors, etc.).

The inputs are for temporary troubleshooting and data is not transmitted to Falcon BMS / DCS. If synchronization is switched on again, manually entered values are overwritten again.

### 3.2.3. Basic Settings

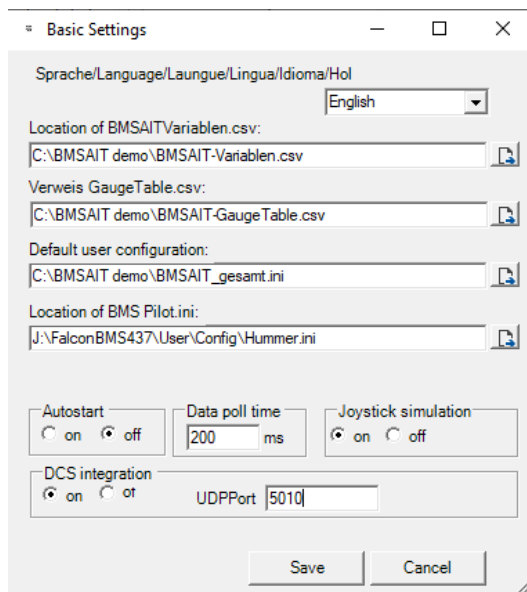


Figure 4: Basic Settings

The software places a configuration file "BMSAIT.cfg" in the folder where the executable file is located, in which the selected basic settings are stored and automatically loaded when the program is started.

The basic settings include:

- Choice of language  
Here you can switch a language for displaying BMSAIT (currently German or English).
- Location of the BMSAIT variable file  
The selection guide can be used to specify the folder in which the file BMSAIT-Variablen.csv is located. This information is mandatory.
- Location of the default user configuration  
This determines which configuration (assignment of Arduino devices, variables, keyboard commands) should be loaded automatically when the program is started. The specification is optional.
- Location of the BMS pilot.ini  
For the output of additional manual frequencies on the backup radio, the location of the Pilot.ini in the User/Config folder of the BMS must be specified here. This information is only required if the backup frequencies are desired (see example BUPRadio).
- Location of the GaugeTable.csv  
Use the selection guide to specify the folder in which the file BMSAIT-GaugeTable.csv is located. This information is mandatory.
- Autostart  
This determines whether to start communication with Falcon BMS and the selected Arduinos immediately when the program starts. The default value is Off.
- Data poll time  
Specifies how long to wait between two queries of current data from the Falcon's SharedMem. Default value is 200ms. When controlling precise instruments (e.g. analogue displays via steppers/servo motors), the value should be lowered.

- Joystick simulation  
If this is activated, BMSAIT connects to one or more virtual joysticks in order to be able to send signals to BMS. This requires that the "vjoy" software has been installed on the computer and that at least one virtual joystick has been set up (see chapter 3.3).
- DCS Integration  
If this option is activated, BMSAIT can also be used to read and transmit flight information from DCS (see Chapter 3.4).
- UDP Port  
Here you can specify on which UDP port the flight information from DCS should be received via DCS BIOS.

### 3.2.4. Set up devices

Figure 5: BMSAIT Device Settings

This form is used to set up a new device or edit an already stored device.

In the field Select COM port, select the port through which you want to communicate with the Arduino. The drop-down list offers all known COM ports. If the COM port of an Arduino does not appear here, the USB connection to the Arduino must be disconnected and reconnected. Call up this form again and the COM port should be available.

In the field Baud rate the connection speed for communication between PC and Arduino is set. The value entered here must match the value used in Arduino programming (see chapter 4.2.2).

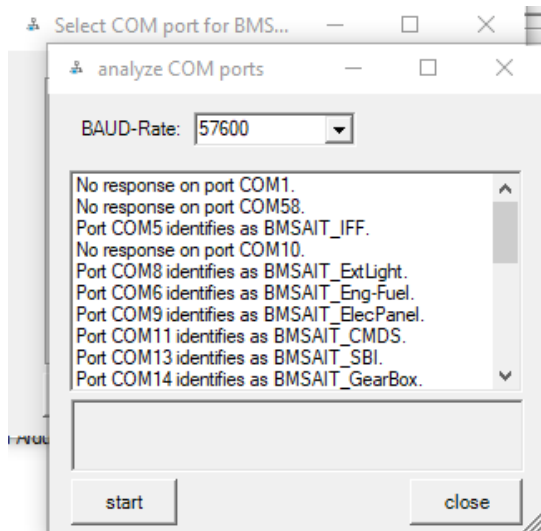
In the field Device Name, a name can be stored for better identification of the Arduino.

The data transfer method specifies the logic of data exchange between the Windows app and the Arduino. (see chapter 2.1.3 and 2.1.4)

The Arduino update rate determines how often data is sent to the Arduino for refresh. The value should be between 50ms and 500ms. Default is 200ms.

The Apply button will perform a plausibility check of the entries and save the changes.

With the Search port button, BMSAIT will contact all COM ports to try to identify any device with the same name as indicated in the field "Device name". If the device is found, the COM port will automatically be selected in the COM port dropdown field.



The button "Show all ports" will call up a new form. Here, the button start will call all COM ports and ask any BMSAIT-configured Arduino to reply with their ID (see 4.2.2 "ID").

### 3.2.5. Variable Selection Form

**selected device**

COM-port: 2 COM11

**selected variables**

ID	type	desc.	
150	float	ChaffCo...	
160	float	FlareCo...	
1547	bool	go	
1548	bool	noGo	

**list of available variables**

ID	group	type	desc.	description
1170	CMDS/R...	float[]	lethality	???
1180	CMDS/R...	int	rwrObjectCount	???
1190	CMDS/R...	int[]	rwrSymbol	???
1200	CMDS/R...	uint[]	missileActivity	???
1210	CMDS/R...	uint[]	missileLaunch	???
1220	CMDS/R...	uint[]	newDetection	???
1230	CMDS/R...	uint[]	selected	???
1240	SYSTEMS	uint	powerBits	Ownship power bus / generator states, see PowerBits
1241	SYSTEMS	bool	busPowerBattery	True if at least the battery bus is powered
1242	SYSTEMS	bool	busPowerEmergency	True if at least the emergency bus is powered
1243	SYSTEMS	bool	busPowerEssential	True if at least the essential bus is powered
1244	SYSTEMS	bool	busPowerNonEssent	True if at least the non-essential bus is powered

**format string:** <@> **test data:** True **id of this data in Arduino:** 6 **pause between data polls (s):** 1

Figure 7: BMSAIT Variable Selection

The form is accessed via the main form if the command "Add variable" is selected in the user configuration by right-clicking on a COM port or the command "Edit variable" for a variable.



The data variable edit form is divided into the following sections:

#### Section 1 – Reference data

This area displays the device/COM port that is currently selected on the main form. It also displays a list of variables that are already associated with the device.

#### Section 2

The Apply button adds the inputs made to a new or existing data variable. Several plausibility checks are carried out beforehand to prevent incorrect entries. After the save is complete, the form will be closed.

The Cancel button does not save any changes and closes the form.

#### Section 3 – Selecting a data variable

This window displays all available data variables of the BMSAIT. The variables were read from the external file "BMSAIT-Variablen.csv" when the program was started.

The sorting of the list can be changed by clicking on the column headings.

An element can be selected from the list. If you want to assign several data variables to a device, this must be done in several calls to this form.

A description of the various columns can be found in the appendix to this document in the description of the "Variable" dataset.

#### Section 4 – additional data

Here, additional characteristics to a data variable can be entered. The Arduino ID field is mandatory and it won't be possible to save the data if this is empty. All other fields are optional, but I recommend filling them with meaningful values.

A description of the various fields can be found in the appendix to this document under the description of the data record "Assignment".

### 3.2.6. Form Input Commands

The screenshot shows the 'Input Commands' window. It has a title bar with standard Windows controls. Below the title bar, there are three main areas:

- Area 1 (Left):** A table with columns 'ID' and 'description'. It lists 10 commands, including 'FormKnobStepUp' (ID 18), 'FormKnobStepDown' (ID 19), 'AerialRefuelingStepUp' (ID 20), 'AerialRefuelingStepDo...' (ID 21), and several 'AntiCollLight' entries. Below the table are 'add' and 'delete' buttons.
- Area 2 (Top Right):** Fields for 'Arduino' (set to 'COM8'), 'ExtLight', and 'KommandID' (set to '18').
- Area 3 (Bottom Right):** A detailed configuration for the selected command 'FormKnobStepUp'. It includes a 'description' field, tabs for 'keyboard', 'JoyBtn', and 'JoyAxis', a 'key' dropdown (set to 'Taste6'), checkboxes for 'SHFT', 'ALT', and 'CTRL' (all checked), and an 'action' dropdown (set to 'press+release'). At the bottom are 'save', 'save and close', and 'cancel' buttons.

Figure 8: BMSAIT input commands

This form is accessed via the button in the basic settings on the main form. The Input Commands form can be used to define keyboard or joystick signals. These signals are assigned an identification number (command ID). If the Arduino sends a command ID, the corresponding signal is converted into a keyboard or joystick signal by the Windows app and thus sent to the currently active application.

**Note:** BMSAIT will only send a signal if Falcon BMS, the Alternate Launcher for BMS or DCS are running (2D or 3D). If not, the test mode needs to be activated in order to process input signals.

The form is divided into three sections:

#### Area 1 – Shortlist

ID	description
18	FormKnobStepUp
19	FormKnobStepDown
20	AerialRefuelingStepUp
21	AerialRefuelingStepDo...
1	AntiCollLight - Off
2	AntiCollLight - 1
3	AntiCollLight - 2
4	AntiCollLight - 3
5	AntiCollLight - 4
6	AntiCollLight - A
7	AntiCollLight - B
8	AntiCollLight - C
9	ExtLightMaster - Off
10	ExtLightMaster - All

add delete

The drop-down list shows all commands that are stored for a selected Arduino device in the current user configuration. An existing entry can be selected by clicking on it in the list. The associated information is then displayed on the right side of the form and can be edited there.

The Add button can be used to create a new command.

The Delete button deletes the selected command.

Area 2 – Header data

ID	description
18	FormKnobStepUp
19	FormKnobStepDown

The header data shows the currently selected Arduino device for which the commands are to be managed. The CommandID field displays the identification number of the signal. In the case of a command that has not yet been saved, the value NEW is displayed here until it is saved.

Below it is a field for a verbal description of the command. This serves to better identify the commands within BMSAIT and can be freely assigned.

Area 3 – Mode selection and detailed information

BMSAIT supports both the triggering of keyboard signals and the use of joystick commands or axes. In the mode selection, a determination can be made as to what kind of signal the command should trigger. If joystick processing has been enabled and the vJoy software has been installed (see 3.3), there are three options to choose from. If vJoy has not been enabled, only the keyboard signal option is available.

Keyboard Signals tab

In the **Button** field, a keystroke can be selected from a drop-down list. If the key is to be pressed in combination with a modifier (Shift, Alt or Ctrl), check the corresponding modifier.

The **action** can be used to set the key behavior (press only, release only, press & release).

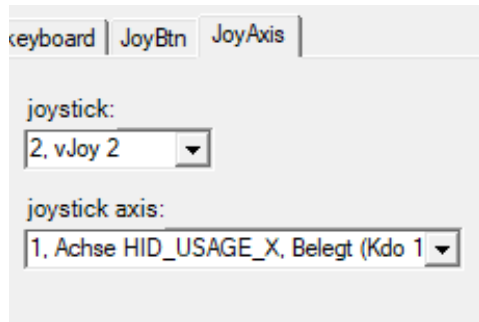
Joystick Button tab

Hint: Only available when vjoy is enabled (see chapter 3.3)

First, select the desired vjoy joystick from the drop-down list in the Joystick field. In the **Joystick Button** field, all available buttons of the joystick are then displayed (the number of buttons can be set in the vJoy software). For each button, it is indicated whether it is still free or has already been assigned

by another command. The desired button must be selected. The **action** can be used to set the key behavior (press only, release only, press & release).

#### Joystick Axis tab

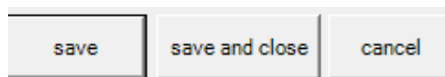


**Hint:** Only available when vjoy is enabled (see chapter 3.3)

First, select the desired vjoy joystick from the drop-down list in the Joystick field. The **Joystick Axis** field then displays all available axes of the joystick (the available axes can be set in the vJoy software). For each axis, it is indicated whether it is still free or has already been occupied by another

command. The desired axis must be selected.

#### Area 4 – Save/Exit



The **save** button saves a change that has just been made. The form remains open for further adjustments.

The **save and close** button applies the change and closes the form.

The **cancel** button discards the current change and closes the form.

**Note:** Saving changes here will only memorize the changes locally. In order to save the changes for the next program start, the user configuration must also be saved (main form click on "Save"/"Save as").

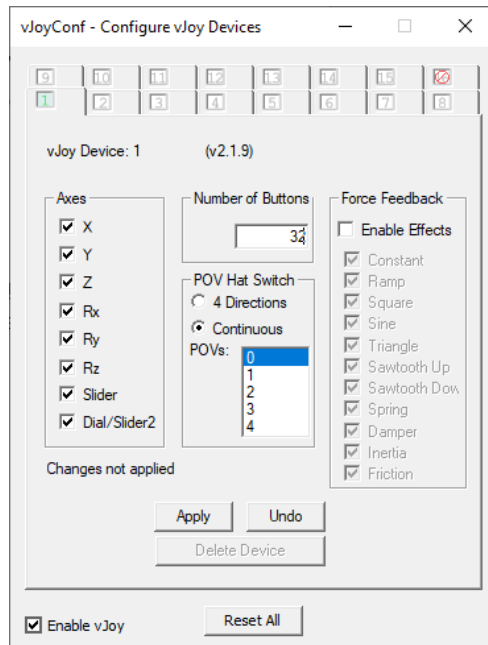
### 3.3.(optional) Activate virtual joysticks

BMSAIT allows you to pass on the switch signals to the flight simulation either as keyboard commands or as joystick signals.

Keyboard commands can InputSimulator.dll be implemented via the library set up. However, if it is desired/necessary to trigger joystick signals (buttons or analog axes) as well, some preparatory work is still required. Joystick signals can only be generated if you have set up a device for this purpose in Windows. BMSAIT can then take over this device and send signals to the flight simulation on behalf of this device.

In order not to have to connect a physical joystick here, there is the option of creating virtual joysticks. For BMSAIT, I use the vJoy software here. You can download this software for free here:

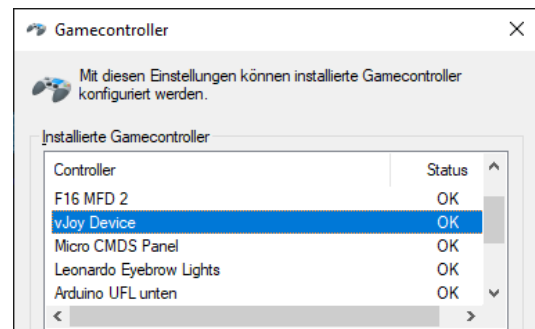
<http://vjoystick.sourceforge.net/site/index.php/download-a-install/download>



After installation, call up the configurator (VJoyConf). In it, you activate a joystick. The standard should be an activated joystick with 32 buttons and all analog axes. ForceFeedback is not required and can be disabled. If you want BMSAIT to control more than 32 buttons or 8 analog axes, you can activate additional joysticks (click on one of the numbered tabs at the top and then click on the "Add Device" button at the bottom).

If you have made changes, Windows must be restarted before they take effect.

Under the game controller setting of Windows, the vJoy should now be visible. If joystick processing is enabled in BMSAIT, it can now be accessed in input processing and an input command of an Arduino can be assigned to the buttons and axes of the vJoy (see 3.2.6).



### 3.4.(work in progress) DCS Integration

As you can see from the name of this project, BMSAIT is an interface for Falcon BMS. Nevertheless, it can happen that you sometimes fly in DCS and don't want to do without the functions of the home cockpit. For Arduino solutions, there are good solutions on both the BMS side (F4toSerial) and the DCS side (DCS BIOS). However, when changing the software, you would also have to adapt the programming of the Arduinos.

With BMSAIT I want to allow the use of both simulations without any need to change configurations. The BMSAIT Windows app automatically registers whether BMS or DCS is started and processes the data received from the simulations to such an extent that no changes to the Arduino programming are required.

This is a lot of work, though. Some basic functions are already functional (e.g. IndexerLights, some engine Instruments), but much more data still has to be checked and compared to such an extent that data is really in identical form and can be displayed correctly by the Arduinos.

To use the function, the "DCS Integration" setting must be activated in the basic settings and a UDP port must be specified on which data is to be received from DCS.

In DCS, an installation of the mod "[DCS-BIOS](#)" (DCS-Skunkwork branch) is required. This software enables the extraction of flight information. The data is sent via the TCP/UDP protocol and received by BMSAIT in this way. DCS BIOS normally transmits on port 5010. In the Export.lua of DCS, an option can be activated that DCS BIOS can also send on other ports. The specific port must be set in the BIOSconfig.lua in the installation folder of DCS BIOS.

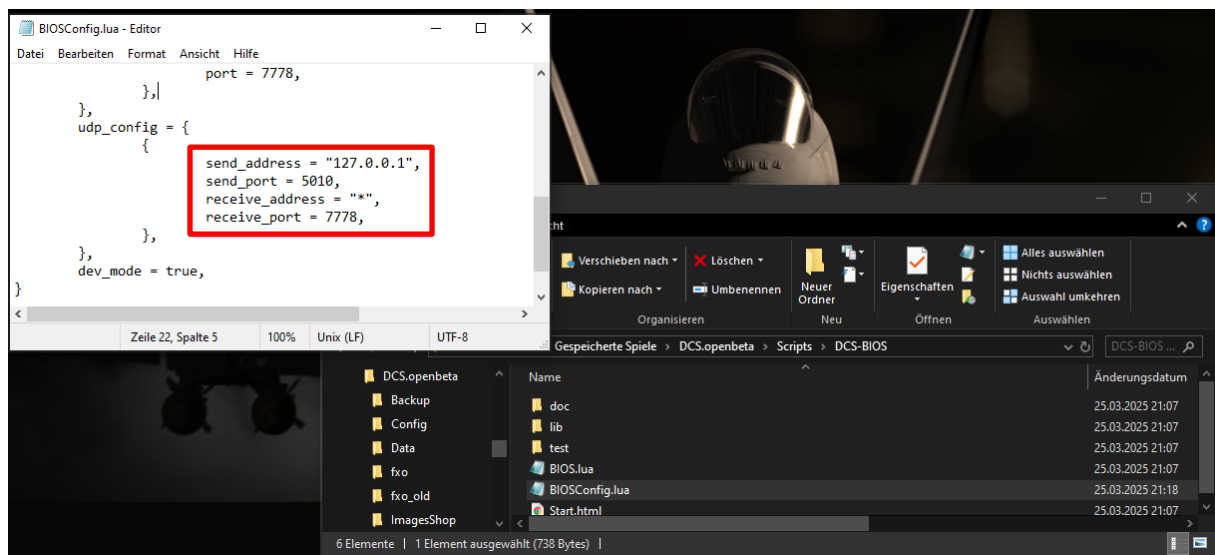


Figure 9: DCS BIOS channel information in BIOSconfig.lua

### 3.5. GaugeTable

In addition to the list of variables, BMSAIT comes with a second CSV file, the "BMSAIT-GaugeTable.csv".

This file contains specifications for controlling analog circular displays (currently RPM and FTIT). The table is necessary because the scale of some displays is not linear, i.e. the distance between two steps is not the same everywhere on the scale (example: On the RPM scale, between 0° (0% RPM) and 90° (50% RPM), 50% of the RPM is covered; between 180° (80%) and 270° (100%) only 20%.

The output from the SharedMem does not take this into account, but only provides the value of the RPM (e.g. 65%) using the RPM display as an example.

The GaugeTable allows the pointer movements to be individually adjusted without having to reprogram the Windows app. In the table, assignments are stored for which values of a scale which angle of the pointer movement is to be achieved. Every user is free to store a few or many assignments here. It is not necessary to specify every value. BMSAIT will interpolate intermediate values based on the "milestones" made.

	A	B	C	D	E	F	G	H	I	J	K
1	<RPM>										
2	Gauge value	60	70	75	80	85	90	100	105	107	110
3	Gauge positi	22500	26000	32250	38500	51000	55750	57250	63500	65535	65535
4											

The screenshot shows the specifications for an RPM scale. The scale is not linear, i.e. the distances between the individual positions (40% -> 50% RPM ; 50% -> 60% RPM) are not constant, but are getting larger and larger. The example with the 65% would move the pointer to position 24250 (corresponds to the motor position 133° clockwise from the zero point) based on these settings of the GaugeTable.

## 4. The Arduino program

### 4.1. Description of the preparations


















#### 4.1.1. Foreword

Arduinos bring with them an almost inexhaustible number of connection options for input and output devices. Because of this, it is not possible to provide a software that covers all possibilities at the touch of a button. The coding presented here therefore does not provide complete solutions, but rather only application examples. When connecting any device, the interaction between Arduino programming/customizing and the wiring of the peripherals must be taken into account. This concerns, for example, the exact assignment of the PINs, a special device or the selection of the right resistors when controlling LEDs.

#### 4.1.2. Programming an Arduino with the BMSAIT Sketch

In order for the Arduino to receive data from the Windows program BMSAIT and output it to connected devices, software must be loaded onto the Arduino. The software is available as C++ program code.

#### 4.1.3. Download the program code

-  BMSAIT\_Vanilla.ino
-  BMSAIT\_UserConfig.h
-  BMSAIT\_Switches.h
-  BMSAIT\_StepperX27.h
-  BMSAIT\_StepperVID.h
-  BMSAIT\_Stepper28BYJ48.h
-  BMSAIT\_SSegTM1637.h
-  BMSAIT\_SSegMAX7219.h
-  BMSAIT\_ServoPWMSHield.h
-  BMSAIT\_Servo.h
-  BMSAIT\_Placeholder.h
-  BMSAIT\_MotorPoti.h
-  BMSAIT\_LED.h
-  BMSAIT\_LCD.h
-  BMSAIT\_Encoder.h
-  BMSAIT\_Buttonmatrix.h
-  BMSAIT\_Analogachse.h

The program code for the BMSAIT functions consists of several files (one .ino file, several .h oder.cpp files). These must be stored together in a folder.

When downloading the BMSAIT software, a collection of all available modules is offered in the "Arduino sketches" folder in the form of various BMSAIT\_ xxxx.h files.


To use BMSAIT, a separate folder should be created for each Arduino controller, in which only the necessary modules that are needed on the controller are stored. But more on that later.

#### 4.1.4. Installation of a development environment

These files contain readable text and can therefore theoretically be viewed with a normal editor. Separate software is required for editing and, above all, for transferring the software to the Arduino. I recommend the free Arduino IDE ([arduino.cc/en/main/software](https://arduino.cc/en/main/software)).

With the Arduino IDE installed, the Arduino software can be accessed by double-clicking on the .ino file.





```

//BASTSETINSTELLUNGEN
#include <Arduino.h>
#include <HardwareSerial.h>

#define DATENLAENGE 8 // Maximale Länge einer Datenvariable (ggf anpassen!!!)
#define BAUDRATE 57600

//Definition der Struktur für den Datencontainer für die Ausgabe von Daten
typedef struct
{
  char bezeichnung[6]; //kurze Bezeichnung der Variable (max. 5 Zeichen)
  char wert[DATENLAENGE]; //Feld für das Speichern des Datensatzes
  int typ; //Angabe, wohin der Wert ausgegeben werden soll (10: LED, 20: L
  int ziel; //Angabe PIN, auf dem der Wert ausgegeben werden soll (Nur LED
  int stellen; //Anzahl der Ziffern für den Datenwert für Formatierungszwecke
  int start; //Position in einer Reihe, ab der der Wert angezeigt werden soll
  int dp; //Angabe, wo der Decimal Point gesetzt werden soll (7Segment)
} Datenfeld;

//Definition der Struktur für den Datencontainer für die Ausgabe von Daten
typedef struct
{
  short nTIN; //Anzahl PIN an dem der Schalter angeschlossen ist
  <

```

When loading, please make sure that the various .h / .cpp files appear as tabs. If not, the files were not stored together with the ino file. The program will probably not be able to run then. However, only the .h/.cpp files that you activate in your project are needed. Unneeded module files can be safely deleted.

Before using it for the first time, it is necessary to install additional libraries (program extensions) in the Arduino IDE. Which libraries are required depends on the devices you want to control via the Arduino.

The following libraries were used in the included examples:

Part	Library
<b>LCD (16x2 or 20x4 displays with HD44067 controller)</b>	LiquidCrystal_I2C.h <a href="https://github.com/johnrickman/LiquidCrystal_I2C">https://github.com/johnrickman/LiquidCrystal_I2C</a>
<b>7-segment display (MAX7219 controller), LED Matrix</b>	LedControl.h <a href="http://wayoda.github.io/LedControl/">http://wayoda.github.io/LedControl/</a>
<b>7-segment display (TM1367 controller)</b>	LEDDisplayDriver.h <a href="http://lygte-info.dk/project/DisplayDriver%20UK.html">http://lygte-info.dk/project/DisplayDriver%20UK.html</a>
<b>Servo motors (direct)</b>	Servo.h <a href="https://github.com/arduino-libraries/Servo">https://github.com/arduino-libraries/Servo</a>
<b>Servo Motor Shield (PCA9685 Controller)</b>	Adafruit_PWMServoDriver.h <a href="https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library">https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library</a>
<b>Stepper Motors (Standard)</b>	Stepper.h <a href="https://github.com/arduino-libraries/Stepper">https://github.com/arduino-libraries/Stepper</a>
<b>Stepper Motors (x27,168)</b>	SwitecX25.h <a href="https://github.com/clearwater/SwitecX25">https://github.com/clearwater/SwitecX25</a>
<b>Stepper Motor Shield (VID6606)</b>	SwitecX12.h <a href="https://github.com/clearwater/SwitecX25">https://github.com/clearwater/SwitecX25</a>
<b>Air Core Motor with Controller (CS4192)</b>	AirCoreCS <a href="https://github.com/alancoolhand/AircoreCS">https://github.com/alancoolhand/AircoreCS</a>
<b>OLED Displays (OLED, FFI, SBI, DED/PFL)</b>	U8g2.h <a href="https://github.com/olikraus/u8g2">https://github.com/olikraus/u8g2</a>

#### 4.1.5. Compilation of the files required to program an Arduino

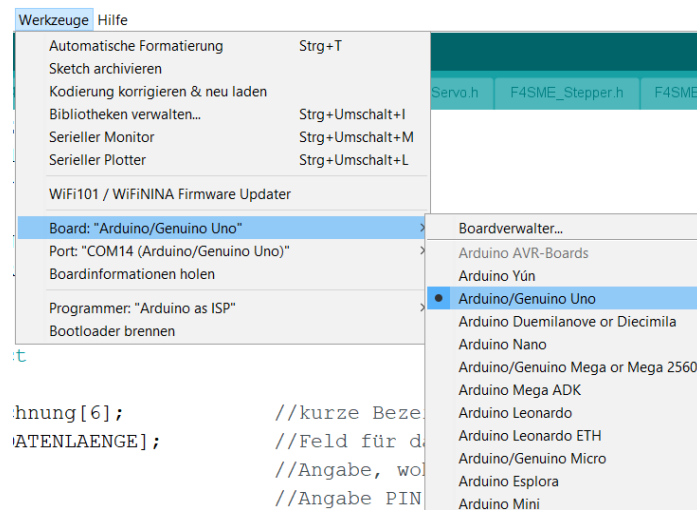
If you use BMSAIT, then I recommend that you create a separate folder for each Arduino controller. Give these folders a descriptive name (e.g. BMSAIT\_ElectricsPanel). Copy a BMSAIT\_XXX.ino (e.g. the BMSAIT\_Vanilla.ino from the folder "Arduino Sketches") into this folder. This .ino file must now be renamed -> the name must match the name of the folder!

In addition to the .ino file, you need the UserConfig.h module, which you can also find in the "Arduino Sketches" folder.

In addition, modules must now be selected that should run on the respective Arduino controller. Copy the desired modules into the folder as well.

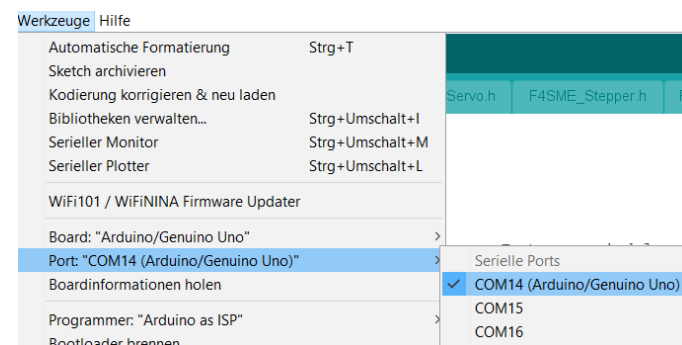
In the UserConfig file and in many modules, settings still need to be made to adapt the software to your individual cabling. What to do is explained in the chapter 4.2 described.

#### 4.1.6. Choosing the Arduino Board



To load the software onto the Arduino, select the Arduino board you want to use in the Arduino IDE. This is done via the Tools tab. In the entry "Board: xxx" the type of board used must be selected (e.g. Arduino UNO).

#### 4.1.7. COM Port Selection



For the "Port" entry, select the serial port that is assigned to the Arduino board in Windows.

The port assigned to a board can be found out via the Windows Device Manager, the Arduino development environment (Arduino IDE) or even in BMSAIT.

#### 4.1.8. Checking the Arduino software

The Arduino IDE checks the software for errors during the upload process. If program errors are found, the loading process is aborted.

I therefore recommend checking the software for program errors yourself before uploading. To do this, select the Control/Compile command in the menu under "Sketch".

The first time this is done, the software may display errors. This is because the program code wants to have access to libraries that are not yet installed (see 4.1.4).

#### 4.1.9. Uploading the Arduino software

In the Arduino IDE, the command Upload can be found in the "Sketch" menu. With this command, if the correct Arduino board / COM port has been selected and the current programming is correct, the software will be loaded onto the Arduino. The Arduino will then restart and begin programmed processing.

### 4.2. Description of the Arduino Sketch

The Arduino software consists of several individual files, which are hereinafter referred to as "modules". The separation into modules serves to clarify the program code, as different application areas (types of devices that are to be controlled via the Arduino) require different programming for control. The division into modules allows the Arduino to load only the program parts that are required for an application. When you downloaded BMSAIT, you were provided with several Arduino projects in addition to the Windows app and this documentation. The BMSAIT Vanilla project forms the basis for this and should be used to start your own projects. The other enclosed projects are examples, each providing a finished, executable project for common use cases. The following description refers to the vanilla project. A description of the example programs can be found in the documentation subfolder at the example program.

#### 4.2.1. The module BMSAIT\_Vanilla

In the main module (the ".ino" file)

1. calls up the required additional modules
2. defines global values and variables (see 5.2.4)
3. Sets up the Arduino (function setup)
4. provides the main loop for continuous processing (Loop function)
5. controls communication with the PC (ReadResponse, SendSysCommand, SendMessage, PullRequest, DebugReadback functions)

To avoid errors, nothing should be adjusted in this module.

Exception: However, an adjustment may be necessary if a new module that is not included in the standard coding is to be mapped (e.g. control of a device that has not yet been taken into account). The new types must be added at the appropriate point in the main loop in order to program the call of a separate function to control the type:

1. The new module must be integrated
2. Reference to the setup of the new module
3. Reference to a calibration of the new module (only for motors)
4. Reference to Endless Loop Update

```

#ifdef ServoMotor
    case 40: //Servos
        Update_Servo(x);
        break;
#endif

#ifdef newDevice //define this flag in the top of F4SME_UserConfig.h to activate this block ("#define newDevice")
    case 69: //assign this type to a variable in the data container to call a new method
        Update_newDevice(x); //program a new method void Update_newDevice(int p){command1;command2;...}to enable your device
        break;
#endif

```

## 4.2.2. The UserConfig module

This module summarizes important settings that the user can/should adjust to set up the desired process. This basic module is required for every project.

### 4.2.2.1. Module Selection

```

4 //MODULE SELECTION - uncomment the modules you want to use
5
6 // #define LED //drive
7 // #define LEDMatrix //drive
8 #define LCD //drive LCD
9 // #define SSegMAX7219 //drive SSegMAX7219
10 // #define SSegTM1637 //drive SSegTM1637
11 // #define ServoMotor //drive
12 // #define ServoPWM //drive
13 // #define StepperBYJ //drive
14 #define StepperX27 //drive StepperX27
15 // #define StepperVID //drive
16 // #define MotorPoti //motorPoti
17 // #define DED_PFL //Enable
18 // #define SpeedBrake //Enable
19 #define Switches //use the switches
20 // #define ButtonMatrix //use the buttons
21 // #define RotEncoder //use the rotEncoder
22 // #define AnalogAxis //use the analogAxis
23 // #define NewDevice //place the new device
24

```

Here it is defined which modules are to be controlled via the Arduino. Only select the modules that will actually be used on this Arduino, as each module takes up resources that may otherwise be lacking for the Arduino's actual work.

To select a module, remove the characters '//' before the word "#define"

To deselect a module, precede the word "#define" with the characters '//'

### 4.2.2.2. Constants (Basic Settings)

```

33 //BASIC SETTINGS
34 #define BAUDRATE 57600 // serial connection speed
35 #define POLLTIME 200 // set time between polls
36 #define PULLTIMEOUT 30 // set time to wait for a response
37 // #define PRIORITIZE_OUTPUT //uncomment this to prioritize output
38 // #define PRIORITIZE_INPUT //uncomment this to prioritize input
39 const char ID[] = "BMSAIT_VANILLA"; //Set the ID for the device

```

#### BAUD RATE

This is where the connection speed is set at which the PC tries to communicate with the Arduino. The specification must match the value specified in the Windows app for this Arduino.

### POLLTIME

This specifies a time in milliseconds that the Arduino waits between two PULL requests. Default is 200ms.

### PULLTIMEOUT

This specifies a time, in milliseconds, that the Arduino waits for a response after a PULL request before continuing processing with the next request.

### Prioritization

Arduinos have very limited computing power and therefore quickly reach the limit of their performance. It can happen that an Arduino reads an incoming message late or not at all, because a data output is being processed that may take a little longer. Conversely, an output device (e.g. a motor) may not be moved evenly because the Arduino is busy reading in the new data in between.

- PRIORITIZE\_OUTPUT enables additional calls to update the data output, thus placing an emphasis on fast output.
- PRIORITIZE\_INPUT activates additional queries from connected switches in order to be able to react to inputs as quickly as possible.

If neither of the two prioritizations is activated, the resources are evenly distributed between the import of fresh data, the evaluation of the inputs and the updating of the outputs.

### ID

Here, the Arduino can be assigned a designation with which the board can identify itself to the Windows app. The default is "BMSAIT\_Vanilla".

#### 4.2.2.3. Device Type (Board Selection)

```
42 //BOARD SELECTION
43
44 #define UNO           //uncommen
45 //#define NANO        //uncomm
46 //#define MICRO       //uncomm
47 //#define LEONARDO    //uncomm
48 //#define MEGA         //uncomm
49 //#define DUE          //uncomm
50 //#define DUE_NATIVE  //uncomm
51 //#define ESP          //uncomm
```

This section selects which type of Arduino is used. The selection is relevant when controlling the OLED displays as well as the adaptation of the communication interface when using the DUE (which has special requirements for communication via the USB port).

#### 4.2.2.4. Data Variables

```

68 Datenfeld datenfeld[] =
69 {
70     //Description ID      DT      OT      target  Ref2 Ref3 Ref4 Ref5  RQ  IV
71     { "RTRIM", "1370", 'f', 60,    0,    0,    0,    0,    0,    0,    "" , "0.0"} //Example
72     , { "PTRIM", "1360", 'f', 60,    1,    0,    0,    0,    0,    0,    "" , "0.0"} //Example
73 };
74 const int VARIABLENANZAHL = sizeof(datenfeld)/sizeof(datenfeld[0]);

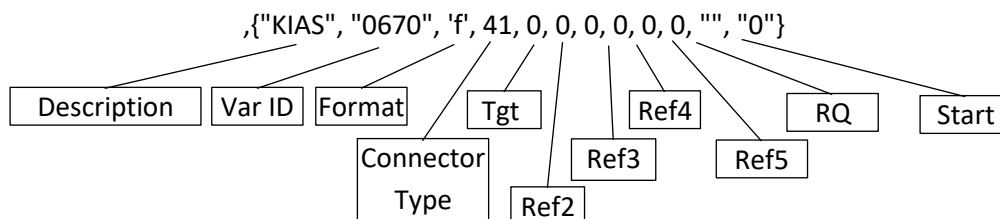
```

Figure 10: Data container

**This is the most important setting that must be made by you.** In this area, it is defined which data of the shared memory of BMS is to be used on the Arduino and which hardware connected to the Arduino displays this data. Theoretically, up to 98 data variables can be added here. However, problems can occur with a large number of variables, as the Arduino's computing capacity is limited and there may be interference or delays in the display. So far, I have not been able to determine where the limit of what is feasible lies.

Important: Please do not overlook the comma at the beginning of each line – only in the first line the comma ist not necessary!

Explanation of the individual positions of a row of the data container:



##### 1. Description

This submission is for informational purposes only. However, the specification can be used to specify not only the value but also the variable designation, e.g. on an LCD display. (max. 5 characters)

##### 2. Variable ID

This entry is only needed if you plan to carry out the data exchange according to the PULL mode. The ID is used by the Windows app to identify the correct record from the SharedMem. Enter the 4-digit number (add leading zeros if necessary) that corresponds to the ID from the variable list BMSAIT-Variablen.csv.

##### 3. Format

This entry is only needed if you plan to carry out the data exchange according to the PULL mode. A key for the data format of the desired data variable, which is also specified in the BMSAIT-Variablen.csv, must be specified.

#### 4. Connector Type

Specifies which connected device to use to display the contents of this data variable.

- 10-LED (PIN is anode and is connected to GND via the LED)
- 11-LED (PIN is cathode and is connected to V cc via the LED)
- 12-LEDMatrix (Many LEDs share the control pins of the Arduino)
- 20-LCD
- 30-7Segment Max7219
- 31-7Segment TM1367
- 32- DotMatrix Module SLx2016
- 40 servo single
- 41 servo via pwm motor shield
- 50-Stepper Motor (28BYJ-48 or equivalent)
- 51-stepper motor (X27.168 directly on the Arduino)
- 52-Stepper Motor (X27.168 via Motor Controller)
- 53-Stepper Motor with Compass Function
- 54- Air Core Motor
- 60 motor potentiometers
- 70-OLED
- 71-OLED with Speedbrake Indicator function
- 72-OLED with Fuel Flow Indicator function
- 80-Control of a backlighting

#### 5. Tgt

The use of the "Target" field depends on the port type.

For connector type 11 (LED), this specifies the PIN to which the LED is connected, on which the (bool) value is to be displayed.

When using engines (type 40,41,50,51,52,54,60), this field is used to create a reference to an entry in the engine list. In the motor lists of the respective modules, control information for each motor can be found (see 4.2.13).

Note: Please be careful not to enter the PIN of the motor here. This information belongs in the motor list of the corresponding module.

When displayed on an LCD screen, this value specifies the line in which the value should be written.

#### 6. Reference 2

The use of this field depends on the connector type.

When controlling an LED, for example, the luminosity of the LED.

#### 7. Reference 3

The use of this field depends on the connector type.

Here, for example, for LCD, you can specify how many characters of the data record are to be displayed. This ensures, for example, on LCD displays or in the case of 7-segment tubes, the correct positioning of the data set. If the record is longer than the number x specified here, only the first x characters of the data variable are displayed.

#### 8. Reference 4

The use of this field depends on the connector type.

Here, for example, determines whether the dataset appears left-aligned or indented on an LCD display or a 7-segment tube. The number given here moves the text on the output device x places to the right.

#### 9. Reference 5

This allows you to specify that a decimal point is set on 7-segment tubes at the location specified here.

#### 10. RQ

This is a placeholder in which the Arduinos prepare the command of a data request using the PULL principle.

#### 11. Seed

The value entered here indicates the default value that should be displayed when the Arduino is turned on.

### 4.2.3. The module Switches

When this module is activated and switches have been defined in the configuration, it continuously checks whether buttons/switches are actuated. When this happens, commands are sent to the PC. In the Windows app, you can define which keyboard/joystick signals should be triggered when a command is present (see 3.2.6).

The programming of this module assumes that all switches are connected to the Arduino PIN to GND.

Two types of switches can be set up in the module:

#### 4.2.3.1. Digital buttons/toggle switches

Any number of switches can be connected to the Arduino (limited by the number of PINs of the Arduino).

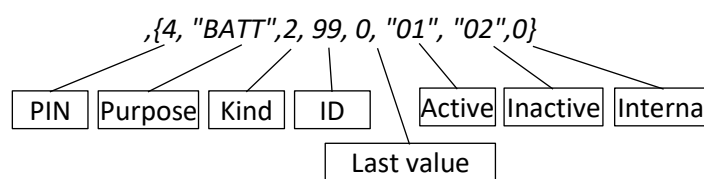
In the data block, a row must be set up for each switch to be read.

```

28//Switch definition. If you add a switch, add a line to the following list
29Switch switches[]=
30 {
31 // <PIN>,<description>,<type>,<rotarySwitchID>, 0, <commandID when pressed>,<commandID when released>,<internal command>
32 {4, "BUP", 2, 0, 0, "01", "02", 0} // DigitalBUP - Backup / Off
33 {5, "AFLAP", 2, 0, 0, "03", "04", 0} // AltFlaps - Extend / Norm

```

Format of a line for the switch:

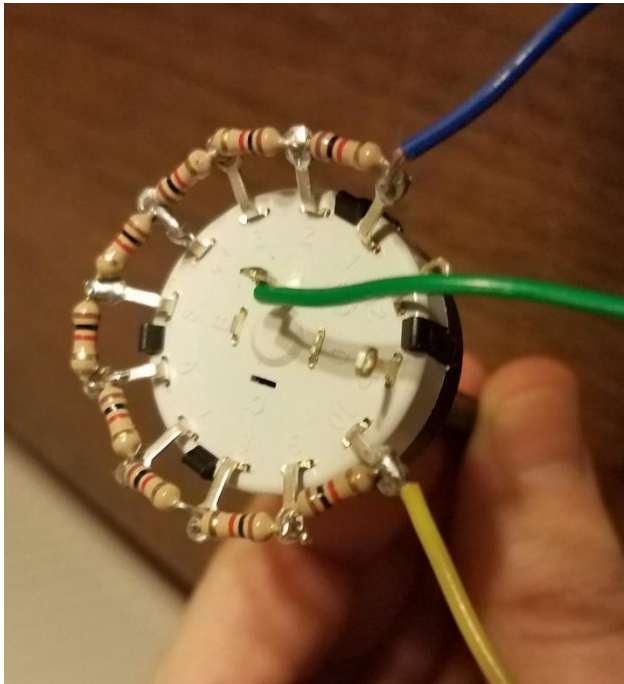




Explanation of the individual items of the line:

1. PIN  
The PIN to which the switch is connected This value is used to identify the PIN on which to search for changed values.
2. Purpose  
A brief description of the purpose of the switch. No special purpose, just for clarity.
3. Kind  
The type of switch. This controls how the switch is read.
  - 1 - digital button,
  - 2 - digital toggle switch,
  - 3 - analogue rotary switch
4. ID  
The identification number is required in order to be able to address the respective command block in the case of several rotary switches that can be read analogue.
5. Last value  
Placeholder for the last measured value (default: 0). Required at runtime. No definition required.
6. Active  
This defines which command is sent to the Windows app when a switch is currently pressed (voltage goes from 1 to 0). Please set "00" here if you do not want to send a command.
7. Inactive  
This defines which command is sent to the Windows app when a switch is released (voltage goes from 0 to 1). Please set "00" here if you do not want to send a command.
8. Internal  
Placeholder for Arduino internal commands. If the key signal is to be used to control Arduino internal things, a value can be stored here that can be processed later. (default: 0) (this is applied in the BUPRadio complex example). The range 1..199 can be assigned for Arduino internal purposes. The number range 200.255 is reserved for communication with the Windows app (see chapter 5.2.5).

#### 4.2.3.2. Digital rotary switches with analogue evaluation



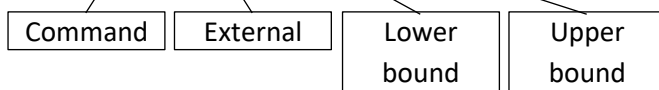
This is a special feature. In order to read a digital rotary switch via an Arduino, each individual detent of the rotary switch is usually connected to a PIN of the Arduino. This can quickly use up all the available PINs of an Arduino. The procedure described here opens up a possibility to read out a rotary switch not digitally, but analogue, in order to control rotary switches with any number of detents via only a single (analogue!) PIN of the Arduino.

If the rotary switch is wired as shown, the movement of the rotary switch influences how many resistors a control signal (green) separates from the power supply (yellow). This results in different voltages that can be read and distinguished by an analog port of

the Arduino. Depending on the measured voltage, it can be used to determine the current position of the rotary switch. The measured value is output by the Arduino in the form of a number (0..1024).

For this to work, a control table must be created for each connected rotary switch according to the following model:

```
//define commands for analog readings of a rotary switch or poti. This is an example for a 10-position switch
//{<CommandID>,<externalCommand>,<low threshold>,<high threshold>}
DrehSchalter analogSchalter[1][12]=
{
  {
    {"04",true,0,64}      //send command 04 if analog read is between 0 and 64 (of 1024)
    ,{"05",true,65,177}  //send command 05 if analog read is between 65 and 177 (of 1024)
    ,{"06",true,178,291} //send command 06 if analog read is between 178 and 291 (of 1024)
    ,{"07",true,292,405} //send command 07 if analog read is between 292 and 405 (of 1024)
    ,{"08",true,406,518} //send command 08 if analog read is between 406 and 518 (of 1024)
    ,{"09",true,519,632} //send command 09 if analog read is between 519 and 632 (of 1024)
    ,{"10",true,633,746} //send command 10 if analog read is between 633 and 746 (of 1024)
    ,{"11",true,747,859} //send command 11 if analog read is between 747 and 859 (of 1024)
    ,{"12",true,860,973} //send command 12 if analog read is between 860 and 973 (of 1024)
    ,{"13",true,974,1024} //send command 13 if analog read is between 974 and 1024 (of 1024)
  }
};
```



##### 1. Command

This defines which command is sent to the Windows app when the current detent is taken (analog value is between the lower and upper bounds of this line). Please set "00" here if you do not want to send a command.

2. External

Here, "true" must be specified if the previously defined command is to be sent to the Windows app when the current detent is taken. If "false" is set here, the command will not be sent to the Windows app. This can be useful if the signal is only to be processed within the Arduino.

3. Lower bound

If a switch is read analogously, a value between 0 (voltage at the PIN corresponds to the supply voltage) and 1024 (voltage at the PIN corresponds to the ground) is output. In order to allow tolerances, the range between 0 and 1024 must be divided into as many ranges as the rotary switch has detents. Here the lower limit of the tolerance range for the current detent is specified. The value should be exactly adjacent to the upper limit of the previous tolerance range.

4. Upper bound

If a switch is read analogously, a value between 0 (voltage at the PIN corresponds to the supply voltage) and 1024 (voltage at the PIN corresponds to the ground) is output. In order to allow tolerances, the range between 0 and 1024 must be divided into as many ranges as the rotary switch has detents. Here the upper limit of the tolerance range for the current detent is specified. The value should be exactly adjacent to the lower limit of the next tolerance range.

#### 4.2.4. The module ButtonMatrix

If an Arduino is to read digital switches, the Arduino board quickly reaches its limits due to the limited number of available PINs. One solution is that instead of connecting each switch to its own PIN, a matrix is used.<sup>1</sup> The switches each short-circuit a combination of two digital output PINs of the Arduino. For example, if 20 PINs are available, the number of controllable switches/buttons on the Arduino is increased from 20 (direct control) to 100 (matrix arrangement).

If only buttons are connected, the cabling is relatively trivial. The reason for this is that only one contact of the matrix is closed at a time with buttons, so that the pressed button can be clearly identified.

If toggle switches are to be connected in addition to buttons - which leads to several contacts in the matrix being closed at the same time - further precautions are required to correctly determine the activated switches. The precautions consist of the fact that a diode must be installed in front of each button/switch.

The following settings must be made in the Arduino module:

In the module, the PINs are to be defined that are to serve as columns/rows of the switch matrix. Place the PINs of the Arduino between the curly brackets where you want to connect the switches.

---

<sup>1</sup> For more information, see <https://www-user.tu.chemnitz.de/~heha/Mikrocontroller/Tastenmatrix.htm>

```
byte rows[] = {2,3,4,5,6,7,8,9};
const int rowCount = sizeof(rows)/sizeof(rows[0]);

byte cols[] = {10,11,12,14,15};
const int colCount = sizeof(cols)/sizeof(cols[0]);
```

In addition, you have to store a table here in which the signals are specified that the Arduino should send to the BMSAIT Windows app when registering a pressed button/switch. Please note that the number of columns (col) and rows (row) must match the number of PINs that you have stored in the previous step.

```
byte keysignal[colCount][rowCount]=
{
// row1 row2 row3 row4 row5 row6 row7 row8
{ 1, 2, 3, 4, 5, 6, 7, 8, } //col 1
,{ 9, 10, 11, 12, 13, 14, 15, 16, } //col 2
,{ 17, 18, 19, 20, 21, 22, 23, 24, } //col 3
,{ 25, 26, 27, 28, 29, 30, 31, 32, } //col 4
,{ 33, 34, 35, 36, 37, 38, 39, 40, } //col 5
};
```

#### 4.2.5. The module Analog Axis



This module makes it possible to read out a connected potentiometer and map it as the analog axis of a joystick via the BMSAIT Windows app and the vJoy software. This can be used to control an analog axis in BMS (e.g. TRIM, volume control, etc.).

To do this, the analog PINs of the Arduino must be named in the module that are to be read:

```
AAchse analogaxis[] = {
// PIN Command Value
{ A0, 2, 0 }
```

##### 1. PIN

Here you have to enter the PIN at which the signal of the potentiometer is taken. Please note that only the PINs of the Arduino that have an analog evaluation capability (A0, A1... Ax).

##### 2. Command

To control an analog axis in vJoy, a command for the analog axis must be defined in the BMSAIT WinApp. Here you have to enter the assigned command ID so that the analog data is connected to the correct vJoy axis.

##### 3. Value

Used at runtime to store the last read value. No modification required.

The analog reading of a PIN is relatively accurate, but 100% precision is not guaranteed. The Arduino divides the range of a connected potentiometer into 1024 positions. It is possible that the signal on

an analog PIN fluctuates slightly due to environmental influences. In order not to trigger an (unnecessary) signal with every fluctuation, a buffer value is also defined here. A signal is only triggered when the signal exceeds the buffer value. I recommend setting the value between 3 and 5.

```

3
4 #define ATH 3 //Analog Threshold. A change of the analog value will only be
5 // considered as a movement if a change in readings is above the threshold valu
~1

```

If a movement of the potentiometer is registered, the read value is sent to the Windows app for further processing and processed there (see 3.2.6).

#### 4.2.6. The module Encoder

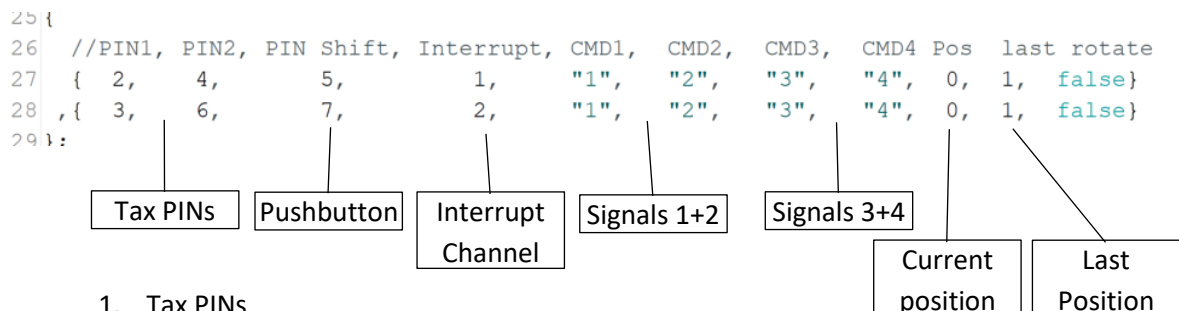
This module enables the control of one or two angular encoders, whose movements are passed on as keyboard or joystick signals. This can be used, for example, to adjust the altimeter or the CRS/HDG knobs of the HSI.

The module is also able to read a push button, which is installed in some angular encoders. This can be used for a "shift" function to trigger four different signals via a rotary encoder.

The angular encoder usually has three PINs. Check your counter's datasheet to determine the PIN assignment. Usually, the middle PIN is to be connected to GND, the two outer PIN to a data PIN of the Arduino. This module works with a special routine of the Arduino, which enables an extremely fast reaction to a movement of the encoder (interrupt) and thus ensures the highest level of reliability. The ability to use interrupts is only available in a few PINs of the Arduino. Each encoder must therefore be connected to one of these PINs.

UN	2, 3
NANO	2, 3
LEONARDO	0, 1, 2, 3, 7
MICRO	0, 1, 2, 3, 7
MEGA	2, 3, 18, 19, 20, 21

If the angular encoder has a probe function, there are two additional PINs on the encoder. If the function is to be used, one contact must be connected to one data PIN of the Arduino and the other to GND.



##### 1. Tax PINs

The two PIN used to read the rotational motion of the angular encoder (the two outer pins of the group of three) must be entered here.

2. PushbuttonPIN

Here you have to enter the PIN to which the button function of the encoder is connected.

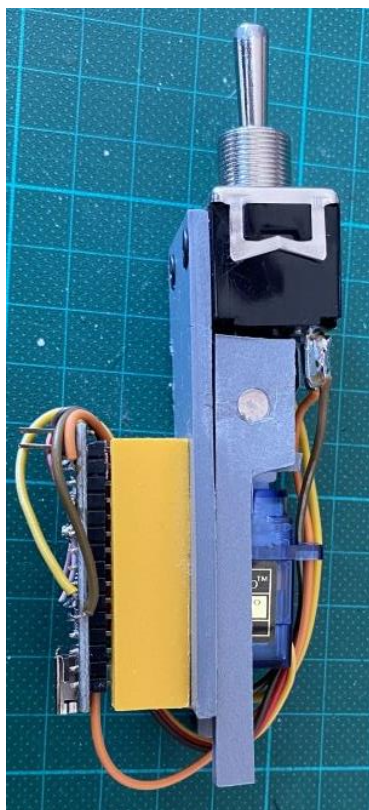
3. Signals 1+2

This defines which command is sent to the BMSAIT Windows app when the encoder moves left/right and the button is not pressed.

4. Signals 3+4

This defines which command is sent to the BMSAIT Windows app when the encoder moves left/right and the button is pressed.

#### 4.2.7. The module Magnetic Switch



In the F-16 there are switches that are held in the knocked-out position when pressed. Either they are released manually or the switch jumps back to the original position at a certain event (e.g. the JFS switch).

Either you buy a real magnetic switch for a lot of money, which enables this behavior, or you use a construction manual to make these switches yourself at low cost.

However, the previous alternatives all have the disadvantage that the switch is held in the rejected position by an active electromagnet. This can lead to complications, as the electromagnets used for this require higher voltages and the Home Cockpit requires a 24V voltage source in addition to the 5V and 12V consumers.

It is possible to control these motors via BMSAIT without any problems: For this purpose, the LED module can be used to send the signal of an active magnetic switch to an electromagnet (note that the electromagnet must not be connected directly to the Arduino, as magnets usually require too high voltages/currents).

The solenoid switch module in BMSAIT therefore takes a different approach: Based on an alternative solution devised by Bumerang of the 47DF, the switch is not held in the deflected position by an active electromagnet, but by a small electromagnet. The Magnetswitch module is used to physically separate the switch from the permanent magnets by a small servo motor and thus bring the switch back to the original position.

The advantages of this approach are the low cost, a small design (only the depth of the switch increases) and the simplification, as no 12V/24V circuits are required.

```
MagSwitchData magSwitchData[] =
{ //dataPos switchPos servoPin servoMin servoMax magSwitchNextStep magSwitchStatus
  { 0, 0, 3, 10, 170, 0, 0},
  { 0, 1, 3, 10, 170, 0, 0}
};
const int magSwitchCount = sizeof(magSwitchData)/sizeof(magSwitchData[0]);
```

The following settings must be made in the Magnetic Switch module:

For each magnetic switch (each switch position is considered a magnetic switch), a row must be created in the magSwitchData data table. The following information must be stored in the data table:

dataPos: The row number in the DataField table of the UserConfig module, which contains the variable that indicates the status of the electromagnet in BMS

SwitchPos: The row number in the switches table where the magnetic switch was defined.

ServoPin: The pin to which the servo motor for the magnetic switch is connected. Important: Not every pin of an Arduino can be used here. It must be a pin that has a ["pulse width modulation"](#) capability.

ServoMin: The minimum value for a deflection of the servo motor, specified as an angle. The value cannot be less than 0 and not greater than the value specified in ServoMax.

ServoMax: The maximum value for a deflection of the servo motor, specified as an angle. The value must not be less than the value specified in ServoMin. The maximum permissible value depends on the design of the servo motor. For the SG90g recommended for this module, this would be 180°.

MagSwitchNextStep: 0, for internal purposes only, don't change anything!

MagSwitchStatus: 0, for internal purposes only, don't change anything!

#### 4.2.8. The module LED



This module contains the function to turn LED on and off. The functions can be used for simple LED solutions where an Arduino is to control a manageable number of LEDs (1 LED per PIN; The current flows through the LED to the ground via an output PIN).

The module processes data variables of types 10 and 11.

Type 10 is to be used if the PIN serves as a power supply for the LED (Arduino PIN → LED → ground).

Type 11 is to be used if the PIN has been connected in such a way that the PIN works as a ground. (Power supply → LED → Arduino PIN).



### Easy control

In the simplest form of control, it is only checked whether the data value assigned to the LED contains the value "T"(rue). If so, the LED will be activated. In all other cases, the LED will be switched off. No other settings are required.

### LED with adapted luminosity

It is possible to set the brightness of an LED in the programming of the Arduino. It is currently not possible to change the brightness via the Windows app.

Arduino UNO:	3, 5, 6, 9 – 11
Arduino MEGA:	2 – 13, 44 – 46
Arduino Leonardo, Micro	3, 5, 6, 9, 10, 11, 13

To determine the light strength, it is necessary to connect the LED to one of the pwm PINs of the Arduino.

The luminosity of the LED is specified in the UserConfig module in the data field. In the row with the data variable that belongs to the LED, in column Ref2, the luminosity must be specified in a range from 0 (off) to 255 (full luminosity).

```

70 Datenfeld datenfeld[] =
71 {
72     //Description ID    DT    OT    target    Ref2 Ref3 Ref4 Ref5 RQ  IV
73     {"RGear", "1599", 'b', 10, 2, 0, , 0, 0, "", "False"} //Example Variable 0 - Right Gear Safe (brightness 0%)
74     , {"NGear", "1597", 'b', 10, 3, 128, , 0, 0, "", "False"} //Example Variable 1 - Nose Gear Safe (brightness 50%)
75     , {"LGear", "1598", 'b', 10, 4, 255, 0, 0, 0, "", "False"} //Example Variable 2 - Left Gear Safe (brightness 100%)
76     , {"UGear", "1571", 'b', 10, 5, 178, , 0, 0, "", "False"} //Example Variable 3 - Gear Unsafe (brightness 75%)
77 };
78 const byte VARIABLENANZAHL = sizeof(datenfeld)/sizeof(datenfeld[0]);

```

### Flashing LED

There are some LEDs in the simulator that flash in certain situations (e.g. the lamp of the JetFuelStarter). The information about whether an LED is on/off and whether it is flashing is stored in different areas in the SharedMem of Falcon BMS. The BMSAIT app puts this information together and transmits a combined state.

This is also the reason why some LEDs can be found in the variable table as a Boolean value and some as a byte value: Flashing LEDs are transmitted by the Windows app as a byte value so that the Arduino recognizes whether the LED is off (0) or on (1) or whether it is flashing slowly (3) or fast (4). The Arduino can then cause the LED to flash decentrally. If the flashing is too fast or too slow, this can be adjusted in the programming of the Arduino via the BLINKSPEED definition in the LED module.

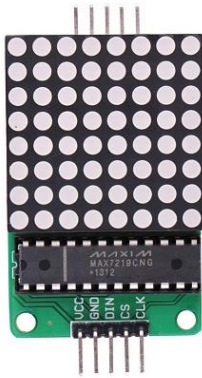
```

7 #define BLINKSPEED 500 //pause (in ms) between on/off for fast blinking. Slow blinking will be 50%

```



#### 4.2.9. The module LED matrix



In this module are the functions to control such a large number of LEDs, where it is no longer possible to connect each LED to its own control PIN due to the limited number of PINs of the Arduino.

The solution is to arrange the LED as a matrix. For this purpose, I recommend using a controller that is cheap to get and greatly simplifies the control. The MAX7219 chip is common here.

The MAX7219 controller is connected to the Arduino via the SPI interface. The LEDs are connected to the output terminals of the MAX7219.

In the module itself, you only have to enter the three PINs with which you connect the Max7219.

```
#define LEDM_CLK 8 //PIN "Clock" for the SPI connection of the LED-Matr
#define LEDM_CS 9 //PIN "Cable Select" for the SPI connection of the I
#define LEDM_DIN 10 //PIN "Data In" for the SPI connection of the LED-Ma
#define LEDM_BRIGHTNESS 5 //sets the intensity of the LED-Matrix
```

With the LEDM\_Brightness setting, you can influence the brightness. The value must be between 0 and 15.

The definition of the individual LEDs is done in the User Configuration module when specifying the corresponding variables.

Ref1 – Number of the MAX7219 chip (default: 1. A different number is only to be specified if several MAX7219 are connected to an Arduino)

Ref2 – Column in which an LED is to be illuminated

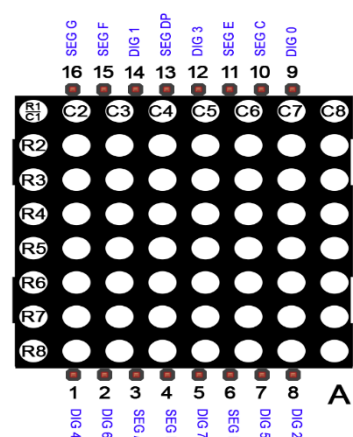
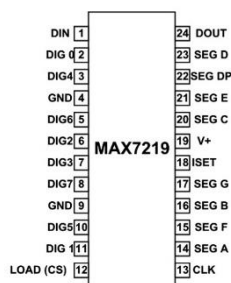
Ref3 – Line in which an LED is to be illuminated

Example variable definition for an LED in an LEDMatrix:

```
{"ENGFI", "1506", 'b', 12, 1, 2, 4, 0, 0, "F"}
```

This variable definition would output the Engine Fire warning light (ID 1506) on the first Max7219 module on the LED connected in the 2nd column and 4th row.

It is quite possible to buy and connect the MAX7219 chip individually. But there are also ready-soldered boards with a MAX7219 chip, which is used to control an 8x8 LED matrix. If you make sure that the module with the 8x8 LED can be easily removed from the board with the controller chip, you can connect your LED directly to the chips without much soldering work.



The designation of the graphic is to be understood as meaning that the PINs of the segments (SEG) control the rows and the PINs of the digits (DIG) the columns. For example, in order to make the 4th LED of the 2nd row light up, a voltage must be applied to the 8x8 matrix between PIN 4 (SegB=2nd row) and PIN 12 (DIG3=4th column).

The LEDs are to be aligned so that the anode (+) is connected to the output connections for the row (Seg) and the cathode (-) to the gaps (Dig).

LEDs can also be made to flash in the LEDMatrix module. As with the LED module, the control works via an entry in the data field of the UserConfig module.

#### 4.2.10. The module LCD



The module can be used to output an alphanumeric value of a data variable to an LCD display. In the present coding, a 16x2 display (HD44780 standard) and control via an I2C module is expected.

In the module, the number of lines and characters per line can be changed if you use a different display size (e.g. 20x4).

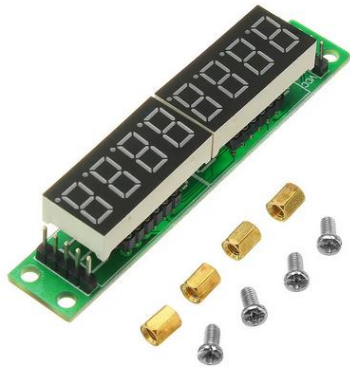
The wiring of the display must be done at the I2C PINs of the respective Arduino board. Since these PINs are predefined in the hardware of the Arduino, it is not necessary to set the PINs in the Arduino sketch.

Board	SDA	SCL
UN	A4	A5
Micro	2	3
Nano	A4	A5
Mega	20	21
Leonardo	2	3
Due	20	21

It is quite possible for several data variables to be displayed simultaneously on an LCD display. For this purpose, the formatting options in the data container must be used, e.g. to display two data variables on the same display in different rows (see 4.2.2 / Target).

It is possible to operate several displays on one Arduino. To do this, remove the comment from the lines of code with the value "lcd[1]" (delete the "//" at the beginning). When operating several I<sup>2C</sup> devices, the hardware addressing of the devices is also important, which must be defined on the LCD display via a solder bridge.

#### 4.2.11. The module SSegMAX7219



With this module, a numerical value of a data variable can be output on an 8-bit 7-segment display tube with SPI connector (MAX7219 chip).

The PIN assignment must be defined in the module. The MAX7219 display is connected to the Arduino via 5 cables. The two cables for the power supply are self-explanatory. Three lines are required for the data connection. Here you have to specify to which PINs of the Arduino the lines for "Clock", "Cable Select" and "Data In" of the display have been connected.

```
#define MAX_CLK 8 //PIN "Clock" for the SPI connection of the 7-Segment Tube
#define MAX_CS 9 //PIN "Cable Select" for the SPI connection of the 7-Segment Tube
#define MAX_DIN 10 //PIN "Data In" for the SPI connection of the 7-Segment Tube
```

The brightness of the display of the 7-segment display can be set when programming the Arduino via the MAX\_BRIGHTNESS option. The value must be between 0 (off) and 15 (full brightness).

It is quite possible that several data variables are displayed on a tube at the same time. For this purpose, the formatting options in the data container must be used, e.g. display a value on the first four characters and another variable on the last four characters (see 4.2.2 / number of characters and starting position).

Note: To set the decimal point (e.g. When displaying a radio frequency), the corresponding location must be specified in the data variable (see 4.2.2 / decimal point).

#### 4.2.12. The module SSegTM1367



With this module, a numerical value of a data variable can be output on a 3 to 6-digit 7-segment display with TM1637 control chip.

Even though the TM1637 is controlled via only two PINs (DIO CLK), it is not via an I2C connection. On the one hand, this means that the connection does not necessarily have to be made to the Arduino's hardware-specified I2C PINs. On

the other hand, it is not possible to operate this display together with other devices via the I2C PINs. It must be specified with which PINs of the Arduino the two data lines of the display were connected.

```
// Call 7-Segment-Display with TM1637 controller
#define TM1637_CLK 2
#define TM1637_DIO 3
```

It is quite possible for several data variables to be displayed on one display at the same time. For this purpose, the formatting options in the data container must be used, e.g. Display a value on the front

two characters and another variable on the back two characters (see 4.2.2 / number of characters and starting position).

Note: To set the decimal point (e.g. radio frequency display), the corresponding location must be specified in the data variable (see 4.2.2 / decimal point).

Example: Control of a 6-digit display via the variable declaration:

Value	Target	Ref3 (Quantity)	Ref4(Offset)	Ref5(point)	Result
Var1: 1234	0	4	0	99	1234__
Var1: 1234	0	4	2	99	__1234
Var1: 1234	0	2	3	1	__1.2__
Var1: 12	0	2	0	1	1.2__34.
Var2: 34	0	2	4	2	

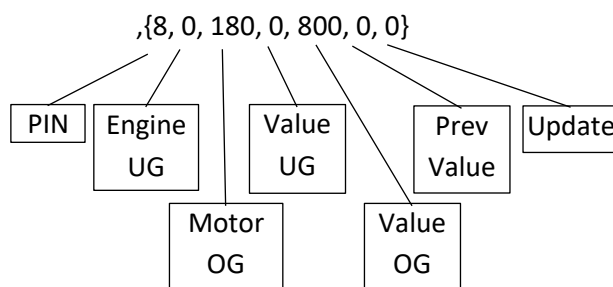
#### 4.2.13. The module Servo



This module is used to control one or more servo motors that are directly connected to the Arduino.

Important: If a data variable is to be output to a motor, the data container of the UserConfig module does not specify the PIN on which the motor is located, but the line in the data container of the Servo module. In addition to the PIN to which the motor is connected, the servo data container also contains other data required for the motor to function.

To control a servo motor, several control pieces of information must be specified. These must be stored in the data container in the Servo module.



##### 5. PIN

The PIN of the Arduino to which the servo motor was connected.

##### 6. Motor UG

The minimum value that the servo should assume (0-180°).

##### 7. Motor OG

The maximum value that the servo should assume (0-180°).

8. Value UG

The lowest value that the data variable can take (number).

9. Value OG

The highest value that the data variable can take (number).

10. Vorh. Value

Placeholder for the last measured value (default: 0). Required at runtime. No definition required.

11. Update

Placeholder for the time of the last control signal to the servo (default: 0). No definition required.

Every motor is different and there are manufacturing tolerances even with identical servos, which is why calibration is necessary. When calibrating the Arduino (see 3.2.1), the engine will move to its minimum position and maximum position and remain there for one second each. During this time, check whether the Arduino vibrates there in a tangible or audible way. If so, then the servo is trying to take a position that cannot be reached due to the hardware limitations. This will probably not damage the servo, but should be avoided in principle. The values Motor\_UG and Motor\_OG must therefore be checked for each connected motor and adjusted if necessary.

Controlling a motor via Arduino is basically no problem. However, synchronizing the movements with a flight instrument from the flight simulator is a bigger task. Servo motors have the disadvantage that the motor cannot move freely, but only has a small range of motion for an arm/pointer (usually 180°). The huge advantage of the servo, however, is that the arm/pointer can be controlled absolutely, i.e. you don't have to know the previous position of the pointer to bring it into a desired position (→ move to position 90°).

The operation of the Servo module is to convert the current value of the data variable into the position that the motor is to control. To do this, the data variable is compared with the values "Value UG" and "Value OG". If the value of the data variable is equal to the defined minimum value "value UG", this means that the motor also moves to its minimum position (angle "motor UG"). If the value of the data variable is exactly in the middle between the minimum and maximum value, the motor will also drive to the middle between the minimum and maximum value.

#### 4.2.14. The module ServoPWMShield

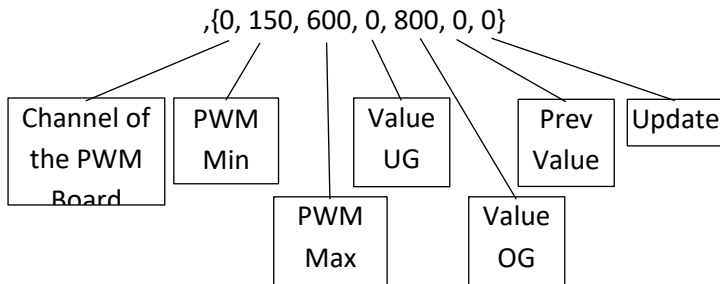


This module is used to control one or more servo motors that are connected to the Arduino via a motor shield (e.g. PCA9684, see left).

To control a servo motor, several control pieces of information must be specified. These must be stored in the data container in the Servo module.

Important: If a data variable is to be output to a motor, the data container of the UserConfig module does not specify the PIN on which the motor is located, but the line in the data container of the Servo module. In addition to the PIN to which the motor is connected, the servo data container also contains other data required for the motor to function.

To control a servo motor, several control pieces of information must be specified. These must be stored in the data container in the ServoPWMShield module.



1. Channel  
The channel on the motor shield on which the servo motor was connected.
2. PWM Min  
The pulse duration that drives the motor to the lower limit (is about 150).
3. PWM Max  
The pulse duration that drives the motor to the upper limit (is about 600).
4. Value UG  
The lowest value that the data variable can take (number).
5. Value OG  
The highest value that the data variable can take (number).
6. Vorh. Value  
Placeholder for the last measured value (default: 0). Required at runtime. No definition required.
7. Update  
Placeholder for the time of the last control signal to the servo (default: 0). No definition required.

Every motor is different and there are manufacturing tolerances even with identical servos, which is why calibration is necessary. When initializing the Arduino, the motor will move to its minimum and maximum position and remain there for one second at a time. During this time, check whether the Arduino vibrates there in a tangible or audible way. If so, then the servo is trying to take a position that cannot be reached due to the hardware limitations. This will probably not damage the servo, but should be avoided in principle. The PWM Min and PWM Max values must therefore be checked and adjusted if necessary for each connected motor.

Servo motors have the disadvantage that the motor cannot move freely, but only has a small range of motion for an arm/pointer (usually 180°). The huge advantage of the servo, however, is that the

arm/pointer can be controlled absolutely, i.e. you don't have to know the previous position of the pointer to bring it into a desired position (move → to position 90°).

The operation of the Servo module is to convert the current value of the data variable into the position that the motor is to control. To do this, the data variable is compared with the values "Value UG" and "Value OG". If the value of the data variable is equal to the defined minimum value "value UG", this means that the motor also moves to its minimum position (pulse duration PWM Min). If the value of the data variable is exactly in the middle between the minimum and maximum value, the motor will also drive to the middle between the minimum and maximum value.

The motor can be damaged if you try to drive it into positions that are outside the permissible angle range (usually 180°). Therefore, please make sure that the stored control information is chosen in such a way that this cannot happen.

#### 4.2.15. The module Stepper



This module is used to control one or more stepper motors (e.g. 28BYJ-48), each of which is connected to the Arduino via a motor control shield (the ULN2003 on the 28BYJ-48). Stepper motors have the advantage that the motor can be moved freely without reaching physical limits. The disadvantage of the stepper, however, is that the motor/software does not know where the pointer is when the program is started. Since the control is relative (→move 20 steps to the right), the previous position of the pointer must be known in order to bring it to a

desired position. This means that a stepper must be calibrated manually or automatically when the program is started in order to then move the arm/pointer into the correct position.

If the stepper is used for a display that has boundaries (e.g. Speed Indicator), the position can be determined by triggering microswitches at the boundaries of the pointer. As soon as the switch is triggered, the position of the pointer is known and can be reliably calculated from there.

For low-torque stepper motors, another option is to simply limit the movement (block the display needle with a lock at the lower limit) and let the motor complete a full rotation. Certain stepper motors (e.g. x27-168) already have these locks built-in.

When implementing displays where a needle must be able to rotate freely (e.g. altimeter, compass), the display must either be set manually or a light barrier must be used that is only triggered at a certain position of the display.

Currently, there is no working example in the delivery of the BMSAIT that demonstrates the functionality of a stepper motor in interaction with flight information from BMS via BMSAIT. I will provide this in a later version.



#### 4.2.16. The module StepperX27

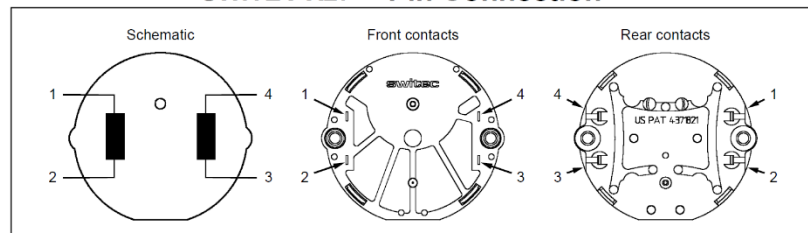


This module is used to control the Stepper Motor X27.168. The special advantage of these motors is that they cannot rotate freely (360°), but only cover an angle of 315°. However, this is not a disadvantage, as these hardware limitations, combined with the low torque of these motors, allow the motor to be brought into a defined state by deliberately "over-revving" at the stops. In this way, the software can find out the position of the advertising nobility and then position it correctly. The X27 can therefore be considered almost like a servo motor, but has the advantage of a longer angular range compared to the usual servos, which can only reliably cover 180°.

The module X27 expects the BMSAIT app to provide a data set that represents the

current position of the motor in the form of a number from 0 (full deflection left) to 65535 (full deflection right). The raw data from FalconBMS must be transposed by the WindowsApp for this purpose. The app also performs non-linear conversion (e.g. display, scale, FTIT or RPM) (see 0).

#### SWITEC X27 Pin Connection



Settings in the StepperdataX27 module:

```
StepperdataX27 stepperdataX27[] =
{
  // {PIN1 PIN2 PIN3 PIN4}   arc   last
  { { 2, 3, 4, 5 }, 315*3 , 0 } // example: FTIT
};
```

Tax PINs

Arc

Last value

##### 1. Tax PINs

The four PINs to which the motor is connected must be specified here. Pay attention to the correct order of the cables with the connections on the motor. If there are errors here, the engine will not behave as desired.

##### 2. Arc

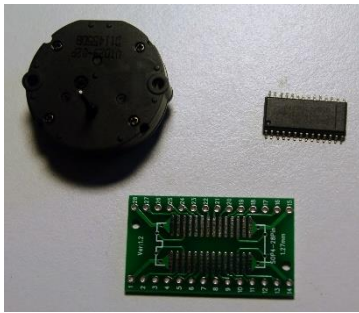
The number of steps that the engine can take between the limits must be indicated here. The accuracy of the X27.168 is 3 steps per degree and an angle of 315°. If the motor is to display a display with a lower angle than 315°, this can be done by reducing the number.

##### 3. Last value

This is where the last known raw data value is stored. This is only needed at runtime, the value does not need to be changed.



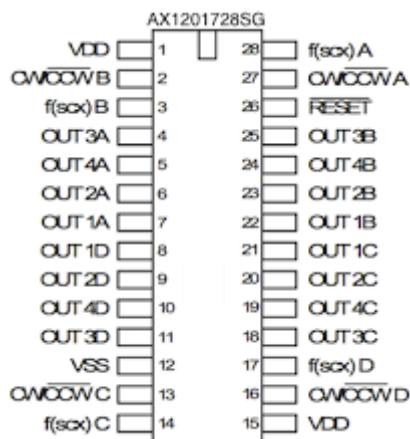
#### 4.2.17. The module StepperVID



This module is used to control several stepper motors of type X27.168 or identical variants (e.g. X40 with two concentric axes). The special advantage of these motors is that they cannot rotate freely (360°), but only cover an angle of 315°. However, this is not a disadvantage, as these hardware limitations, combined with the low torque of these motors, allow the motor to be brought into a defined state thanks to the limitations (the software must know the position of the motor in order to move a display to the correct position). The X27 can therefore be considered almost like a servo motor, but has

the advantage of a longer angular range compared to the usual servos, which can only reliably cover 180°.

Each X27.168 requires four PINs for control, which may exceed the capacity of an Arduino. In addition, the power supply of an Arduino is not sufficient to reliably control several motors.



This is where the controller Chip VID6606 comes into play. One chip can control up to four motors and relieves the Arduino with lower PIN requirements (only 2 per motor) and takes over the power supply of the motors.

The PIN assignment of a VID6606 is shown on the right.

The StepperVID module expects the BMSAIT WinApp to produce a data set that represents the current position of the motor in the form of a number from 0 (full deflection left) to 65535 (full deflection right). The raw data from FalconBMS must be transposed by the WindowsApp for this purpose. A non-linear conversion (e.g. display, scale, FTIT or RPM) is also carried out by the WinApp.

Settings must be made in the Arduino coding of the module.

```
--
{
  // {PIN Step PIN Dir}      arc      last
  { { 8, 9 }, 315*12, 0 }, // exan
  { { 6, 7 }, 315*12, 0 }, // exan
  { { 4, 5 }, 225*12, 0 }, // exan
  { { 2, 3 }, 315*12, 0 }  // exan
}
```

Tax PINs

Arc

Last value

##### 1. Tax PINs

The two PINs used to transmit the control signals for a motor to the VID6606 must be specified here. The first PIN controls the desired direction of rotation of the motor and the second PIN gives the command to move.

##### 2. Arc

The number of steps that the engine can take between the limits must be indicated here. The VID6606 enables an accuracy of 12 steps per degree with an angle coverage of 315° of the

motor. If the motor is to display a display with a lower angle than 315°, this can be done by reducing the number.

### 3. Last value

This is where the last known raw data value is stored. This is only needed at runtime, the value does not need to be changed.

## 4.2.18. The module Air Core



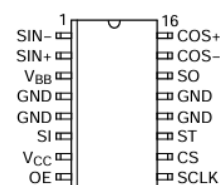
This module is used to control Air Core motors. The difference between these motors and stepper motors lies in the analogue instead of digital control. While stepper motors require a series of control pulses to move the motor, these motors are controlled by a constant analog signal. The advantage of this type of motor is that the analog signal does not send a relative signal (move 10 steps to the right from the current position), but an absolute signal (move to the 90° deflection). With these motors, it is therefore not necessary to determine a zero point before use, as is the case with normal stepper motors.

Note: Air Core motors do not have a stop that restricts the direction of rotation. Therefore, these motors can theoretically also be used for freely rotating instruments (e.g. analog altimeter). However, the software library does not currently allow this – if the instrument is to rotate from 359° to 001°, the motor will not do this in a 2° clockwise step, but with a 358° counterclockwise rotation. I haven't had time to look at the library yet and check if I can get it reprogrammed for BMSAIT.

This module requires the Air Core motor to be connected via a CS4192 controller chip. This chip makes it much easier to control the motor.

The AirCore module expects a data set from the BMSAIT WinApp that represents the current position of the motor in the form of a number from 0 (full deflection left) to 65535 (full deflection right). The raw data from FalconBMS must be transposed by the Windows app. A non-linear conversion (e.g. display, scale, FTIT or RPM) is also carried out by the app.

### PIN CONNECTIONS



Settings in the Arduino sketches:

If a data variable is to be output to an Air Core motor, the data container of the UserConfig module does not specify the CableSelect (CS) PIN on which the CS4192 controller is located, but the line in the data container (0..x) of the AirCore module. In addition to the CS-PIN to which the motor controller is connected, the AirCore data container also contains other data required for the motor to function.

In the AirCore module, two settings must be made.

1. In the `aircoreData` table, the information to which PIN of the Arduino the Cable-Select PIN of the CS4192 controller is connected must be entered for each connected Air

```
AirCoreData aircoreData[] = //fill this table with the specific data
{
  // PIN   arc   last
  { 10, 1023, 0 } // example: RPM
  //,{ 9, 1023, 0 } // example: FTIT
};
const int aircoreNum = sizeof(aircoreData)/sizeof(aircoreData[0]);
```

Core. The entry "arc" indicates the maximum deflection of the Air Core (these motors are available in 360° (4 PINs) and 180° (3 PINs) variants. In the case of a 360° deflection, the value "arc" is 1023. At the 180°, the value 511 must be entered. The value "last" is used internally and should be left at 0.

- In the aircoreMotor table, a placeholder must be inserted for each Air Core connected to the Arduino. So are e.g. three Air Core motors connected, the line AirCoreCS(13) must appear three times (separated by commas) in the table.

```
AircoreCS aircoreMotor[aircoreNum]=
{
  AircoreCS(13)
  //,AirCoreCS(13) //uncomment if you want to dive multiple motors on one Arduino.
};
```

#### 4.2.19. The module MotorPoti



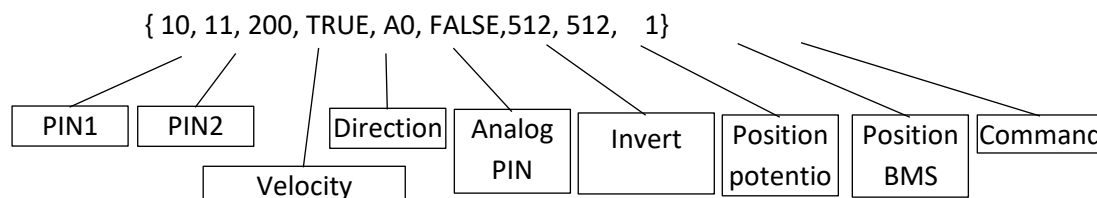
This module is used to control one or more potentiometers, where an analog value can be read out and passed on, but the position can also be changed by a connected motor (example: controller on the trim panel).

The software continuously compares the value of the potentiometer, the position of an analog axis in BMS and the effects in Falcon BMS (trim value). If a deviation is detected, it is determined whether the value has been changed in the simulation or whether the potentiometer has been rotated. The other value is then adjusted.



The motor of the MotorPoti must be connected to the Arduino via a MotorShield (H-bridge). I recommend a controller board like the HG7881 (see left).

In the MotorPoti module, a configuration of the motor potentiometer must be made. For each motor potentiometer, a line must be added to the block "Struc\_MotorPoti motorPoti[]={}".



- PIN1

The PIN that indicates the direction of rotation of the motor (A1, B1, C1, or D1 of the HG7881 board)

- Pin2

The PIN used to trigger a rotary command for the motor (A2, B2, C2 or D2 of the HG7881 board)

3. Velocity

Here a pulse duration is specified (0..255), which is used to control the speed at which the motor should move. This only works if the motor potentiometer is connected to the Arduino's PWM pins suitable for this purpose (example Uno/Nano: 3, 5, 6, 9, 10, 11). If non-PWM pins are to be used here, a value of at least 124 must be specified for the speed.

4. Direction

This is where the last direction of rotation of the motor is stored.

5. Analog PIN

The PIN to which the potentiometer's Signal PIN is connected must be entered here.

6. Inverting

Default is FALSE. If the analog axis has been inverted in BMS, TRUE must be entered here.

7. Position potentiometer

This is where the value that the potentiometer last assumed is stored (0..1024).

8. Position BMS

This is where the value that is to be assumed according to the simulation is stored (0..1024).

9. Command

Here you have to specify the command command that WinApp uses to link the potentiometer to an analog axis of the vJoy (see 3.2.6, Joystick Axis group).

The Engine Potentiometer module automatically tries to synchronize the direction of motion of the engine, the resulting direction of movement of the potentiometer, the analog axis of the vJoy, and the axis assignment in Falcon BMS. However, if there are problems with this module, it may be necessary to replace the wiring of the motor potentiometer (+/- on the motor or the +/- on the potentiometer).

#### 4.2.20. The module OLED

The OLED module is intended to display simple information on an OLED.

The biggest difficulty is to achieve the correct control of the display via the Arduino. Since there are many different OLED sizes, communication chips, providers and different connection options, a pre-configuration must be done here. This is done at the beginning of the sketch by defining a constructor that contains all the relevant information required for correct communication.

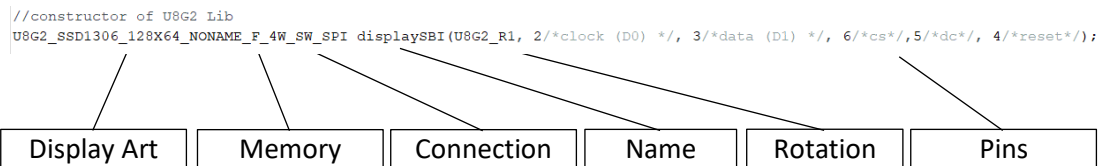
##### The U8G2Lib constructor:

For this and the following modules SpeedbrakeIndicator, FuelFlowIndicator and the DED/PFL, the most important setting is the integration of the OLED.

The OLED display is controlled via the Arduino library U8G2. Since there are many different types and sizes of OLEDs, the library must be given the information of the display used. This is done through the constructor, which is specifically designed for a specific type of display. You can find the full list of constructors here:

<https://github.com/olikraus/u8g2/wiki/u8g2setupcpp>

The constructor consists of a display type, a memory management setting, and a connection type.



### 1. Display Art

From the reference list of constructors, find the entry with your controller chip of the display size and copy it to the corresponding place in the header of the ArduinoCode of the BMSAIT module.

### 2. Memory management

```
/// Declare screen Object
#if defined(DUE) || defined(DUE_NATIVE) || defined(MEGA)
//arduino board with enough memory will use the unbuffered mode
U8G2_SSD1322_NHD_256X64_F_4W_SW_SPI displayDED(U8G2_R0, /* c1
#else
//arduino board with low memory will have to use the buffered
U8G2_SSD1322_NHD_256X64_I_4W_SW_SPI displayDED(U8G2_R0, /* c1
#endif
```

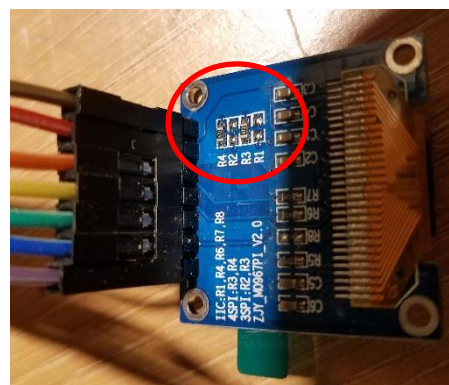
Controlling an OLED can also take up a lot of the Arduino's resources. For this reason, a block is stored in the code that determines how the Arduino controls the display depending on the

Arduino type selected in the user configuration (this applies to the "memory" value described in the following paragraph). If you are using a different display than in my example and need to replace the constructor lines, make sure that the memory management is set to "F" in the first block and "I" in the second block.

### 3. The connection type

Here you can specify how you want to connect the display to the Arduino. Depending on the controller chip, there is the possibility of control via a 2-cable I2C via a 3- or 4-cable SPI to an 8-cable parallel control. I recommend a 4-cable SPI connection if this is offered (4W\_SW\_SPI in this example).

Note that the connection type may have to be set on the board of the OLED display via a solder bridge!



### 4. The name

Here you have to specify the variable name under which the OLED is addressed in the Arduino source code.

Module	Name in constructor
OLED	displayOLED
SBI	displaySBI
FFI	displayFFI
DED/PFL	displayDED

#### 5. The rotation

Here, an option determines whether the display is rotated (U8G2\_R0 = not rotated; U8G2\_R1 = 90° CW ; U8G2\_R2 = 180° ; U8G2\_R3 = 90° CCW).

#### 6. The PINs

Enter the PINs where the display was connected to the Arduino.

The OLED module also has layout customization options. The position of the text to be displayed can be influenced by the following settings:

```
//Layout settings
//Rotation: U8G2_R0 U8G2_R1 U8G2_R2 U8G2_R3
//           (0°)   (90°CW) (180°) (270°CW)
#define OFFSETX 0 //increase this to move Text right down left up
                  //decrease this to move Text left up right down
#define OFFSETY 0 //increase this to move Text down right up left
                  //decrease this to move Text up left down right
```

Normally, increasing the OFFSETX parameter moves the text to the right. However, this only applies if the display has not been rotated (see constructor option for rotating the OLED). The table shown to the right of the option provides guidance on which value needs to be increased/lowered in order to move the text in the desired direction depending on the rotation of the display.

### 4.2.21. The module Speedbrake Indicator



The Speedbrake Indicator module is used to display the Speedbrake status on an OLED display.

This module is largely based on DEDuino (<https://pit.uriba.org/tag/deduino/>) and has only been adapted for data communication with BMSAIT and a few display controls.

Depending on the position of the speedbrake, the text "CLOSE", the graphic for the open speedbrake (3x3 circles) or the OFF display should appear.

Small displays (approx. 1 inch) with 128x64 pixels are suitable for the image. In the coding, I assume that the display is controlled via the very common controller chip SSD1306. Other displays also work, but require an adjustment (see settings for the structure).

#### A) The U8G2 Constructor

The necessary settings are described in the chapter 4.2.20 described.

In addition, enter the number of pixels of your display (default: 128 width and 64 height) in the "SB\_SCREEN\_W" and "SB\_SCREEN\_H" options.

## B) Layout Settings

Basically, setting the correct constructor should be enough to display the SBI. However, in order to be able to control the display precisely, I have stored a value range in the module that can be used for the fine positioning of the display on the OLED.

If the SBI is shifted for you, you can move the display by adjusting the OFFSETX and OFFSETY options to fit exactly into the panel bezels (see chapter 4.2.20).

```
#define SBIDELAY 250  
//#define ANIMATION  
#define OFFSET_FRAMES 6
```

The SBIDELAY option specifies the time, in milliseconds, for how long the OFF display is displayed after the Speedbrake is moved (default: 250).

The ANIMATION option determines whether the Speedbrake Indicator display is animated when the status changes. If the option is disabled, the OFF flag is always shown as a still image when switching between OPEN/CLOSED. If the option is activated, the display scrolls from one mode to another. The OFFSET\_FRAMES option determines how many intermediate steps are displayed. The more steps you set here, the smoother the display will theoretically be. In practice, the display result depends on the performance of the Arduino and the display.

## Module behavior:

- The display is basically deactivated. If BMSAIT is not active on the PC and therefore no data is received from the Arduino, the display will go into sleep mode after 10 seconds.
- If BMSAIT is active, but BMS/DCS is not in the 3D world, the "OFF" indicator will be displayed.
- If the power supply to the machine is not active, the "OFF" indicator is displayed.
- If the animation is deactivated, the "OFF" indicator is briefly displayed whenever the Speedbrake is moved.
- If the Speedbrake is closed (below 1°), the "CLOSED" display is displayed
- If the Speedbrake is open (above 1°), the "OPEN" display is displayed



#### 4.2.22. The module FFI



This module allows the FuelFlowIndicator to be displayed on an OLED display.

In order to be able to control the OLED display correctly, settings are required (constructor). These are divided into chapters 4.2.20 .

A special data variable is required for display, in which the text to be displayed is provided by the BMSAIT Windows app in the correct formatting (BMSAIT variable ID 0511/0521).

This module is largely based on DEDuino (<https://pit.uriba.org/tag/deduino/>) and has only been adapted for data communication with BMSAIT and a few display controls.

##### A) The U8G2 Constructor

The necessary settings are described in the chapter 4.2.20 described.

In addition, enter the number of pixels of your display (default: 128 width and 64 height) in the "SB\_SCREEN\_W" and "SB\_SCREEN\_H" options.

##### B) Font settings

```
// FONT DEFINITIONS - Main
#define ffFont FalconFFI /
#define FF_CHAR_W 20 // -
#define FF_CHAR_H 30 // -
```

This is where the font to be displayed on the display is set. This has been set specifically for the FFI and should not be changed. It should be noted that this font contains only the mandatory letters (numbers, letters for "FuelFlow" and "PPH"). Therefore, other content cannot be displayed on the display.

The FF\_CHAR\_W and FF\_CHAR\_H options tell the software the size of the characters. These should not be changed.

##### C) Layout Settings

```
#define FF_OFFSETX 0
#define FF_OFFSETY 0
```

These options can be used to influence the position of the FuelFlow value on the display, if this is necessary for fine alignment of the text in a panel/bezel (see chapter 4.2.20).

##### D) Options

```
#if defined(MEGA) || defined(DUE) || defined(DUE_NATIVE)
#define REALFFI
#define BEZEL
#endif
```

The FFI display of Uri\_ba has two additional options for displaying the FFI. These options are enabled by default

when using Arduinos with good performance. If this is not desired, these features can be deactivated by commenting out ("//" in front of it).

The REALFFI option animates the digits when the FuelFlow changes.

The BEZEL option draws a frame on the display with the fixed text "Fuel Flow" and "PPH".



#### Module behavior:

- The display is basically turned off. Without connection to BMSAIT, the display is not active.
- If a data connection with BMSAIT is active, but FalconBMS is not in the 3D world, a null value is displayed.
- As long as the PC is in the 3D world, the last valid FuelFlow will be displayed.
- If the PC leaves the 3D world, a null value is displayed again.
- The display goes off once no more data has been received from BMSAIT for 10 seconds.

#### 4.2.23. The module DED/PFL



BMSAIT can also be used to control OLED displays for various purposes. OLED displays are available in many different sizes and different controller chips, so it is not possible to provide a comprehensive solution for all types of OLEDs with this module.

This module is based on DEDuino (<https://pit.uriba.org/tag/dedduino/>), but has been adapted for use with BMSAIT in some places.

##### A) The U8G2 Constructor

The OLED display is controlled via the Arduino library U8G2. In order to call up the library and be able to use the OLED correctly, a constructor must be defined in the head of the module. The settings required for this are described in chapter 4.2.20 described.

##### B) Options

```
#define PRE_BOOT_PAUSE 1000
#define POST_BOOT_PAUSE 1000
```

The way the DED is controlled does not allow for individual layout settings. The only options are therefore the specifications PRE\_BOOT\_PAUSE and POST\_BOOT\_PAUSE, which can be used to specify the time in milliseconds that the DED displays the test image when booting up.

#### Functionality:

This module allows the DED or PFL to be displayed on a 254x64 OLED display.

For the display of the DED and the PFL, 5 lines of 24 characters each are to be displayed. With the display size of 254x64 characters, this results in a character size of 12x10 pixels.

It should be noted that a font must be installed to output text on the OLED. The font has a fixed size for each character. Therefore, you have to know in advance which information should be shown on the display and how large in order to be able to load the corresponding font. Due to the limited resources of an Arduino, you have very limited options to store several fonts. The DED module comes with a font that is stored in the file FalconDEDFont.h as C code (see attachment 5.2.5). The font contains only the characters required to display the DED (lowercase letters and some special characters are missing) and is not suitable for other representations.

Each character is assigned a code that usually corresponds to the ASCII code, but differs for certain DED-specific characters.

The data of the DED are found in two variables of the SharedMem of Falcon BMS (230 "DED" and 245 "INV"). However, this data is not readable without further ado, as the DED has special characters that are not included in the normal character scope (e.g. inverted text, cursor arrows). The BMS Windows app therefore reads the data from the SharedMem and changes the bytecode before sending it to the Arduino. On the Arduino, the correct assignment to a character to be displayed according to the stored value set is already received and can be output directly.

Module behavior:

- The display is basically turned off. Without connection to BMSAIT, the display is not active.
- If a data connection with BMSAIT is active, but FalconBMS is not in the 3D world, a test value is displayed.
- As long as the PC is in the 3D world, the current DED/PFL content will be displayed
- If the PC leaves the 3D world, a test value is displayed again.
- The display goes off once no more data has been received from BMSAIT for 10 seconds.

#### 4.2.24. The module Backlighting

BMSAIT enables software-controlled backlighting of a panel. This makes it possible to dispense with a separate circuit for the panel lighting. It is possible to control single LED or multiple LEDs. However, you have to pay attention to the limitation of the Arduino board. Direct control of the LED is only possible up to 5V and a maximum of 40mA. If this is not sufficient, a relay and an external power supply can be used.

The panel lighting is connected to the Arduino and activated/deactivated by software commands. The software control is linked to the instrument lighting of Falcon BMS by default, i.e. BMSAIT will activate the backlight when the lighting is activated in the simulation.

## 5. Appendix

### 5.1.Sources / references

"F4SharedMem.dll" by LightningViper (<https://github.com/lightningviper/lightningtools>).

Windows Input Simulator Plus by Michael Noonan, Theodoros Chatzigiannakis  
(<https://github.com/TChatzigiannakis/InputSimulatorPlus>).

Virtual Joystick Emulator vJoy by Shaul Eizikovich (<http://vjoystick.sourceforge.net/site/>)

The BMSAIT icon was drawn by Ahmad Taufik and downloaded from the thenounproject.com page .

Code modules for the modules DED, FuelFlow were taken over from the DEDunino project by Uriba  
(<https://pit.uriba.org/tag/deduino/>).

The code for the X27 stepper motors as well as the control of the VID6606 chip come from Guy Carpenter and were taken from the site  
<https://guy.carpenter.id.au/gaugette/2017/04/29/switecx25-quad-driver-tests/> .

DCS BIOS is continued by the DCS Skunkworks (<https://github.com/DCS-Skunkworks/dcs-bios>)

## 5.2.Data field descriptions

### 5.2.1. Data variables (BMSAIT-Variablen.csv, Windows app)

#### ID

The ID is a unique identifier of a data variable. In program coding, the ID is assigned to a range of the BMS's sharedMem. A change or addition to the IDs is therefore not possible without the intervention of the developer.

#### Group

The group is a self-defined division of the data variables into different categories. This is only for sorting.

#### Format

The format (other term: data type) determines what value a variable can take on and what operations are possible with it. SharedMem data is available in different formats. BMSAIT has to proceed differently depending on the data type, which is why specifying the format in the Windows app and Arduino is important. The format/data type of a BMSAIT data variable is fixed in the BMSAIT-Variablen.csv. Specifying an incorrect data type in the Format field can cause problems.

<u>Data type</u>	<u>Term long</u>	<u>Abbreviation</u>	<u>Range</u>
<u>Data bytes</u>	<u>Byte</u>	<u>y</u>	<u>0x00 – 0xFF (decimal: 0-255)</u>
<u>Integer</u>	<u>Integer</u>	<u>i</u>	<u>-32.768 to 32.767</u>
<u>Decimal number</u>	<u>Float</u>	<u>f</u>	<u>-3.4*10<sup>38</sup> to +3.4*10<sup>38</sup></u>
<u>Text</u>	<u>String</u>	<u>s</u>	<u>Any number of characters</u>
<u>Truth value</u>	<u>Bool</u>	<u>b</u>	<u>T(rue) – F(alse) (decimal 0-1)</u>
<u>Data byte group</u>	<u>Bytes[]</u>	<u>1</u>	<u>Any number of bytes</u>
<u>Integer group</u>	<u>Int[]</u>	<u>2</u>	<u>Any number of integers</u>
<u>Text group</u>	<u>String[]</u>	<u>3</u>	<u>Any number of texts</u>
<u>Decimal number group</u>	<u>Float[]</u>	<u>4</u>	<u>Any number of decimal numbers</u>

The BMSAIT-Variablen.csv file also contains variables with the data types ushort, uint, and uint[]. These are currently not supported by BMSAIT. If (contrary to expectations) there is a need to transfer these variables, please contact me.

#### Type

The type is an indication of the data format in which the information of this data variable is stored. The information is used as a processing aid by the BMSAIT and is transmitted to the Arduino with each data packet.

#### Designation

A short description that was usually taken from the F4SharedMem library.

#### Description

A somewhat more detailed description of what information can be found in the data variable. This was usually taken from the F4SharedMem library.

### 5.2.2. Variable mapping (Windows app)

#### Formatting string

The formatting string makes it possible to influence the structure of the data packet that the program sends to the Arduino. It is imperative that the formatting string contains the character '@'. This character represents the data value to be transferred as a placeholder. All characters that precede this character are preceded by the data variable when it is transmitted. Similarly, this applies to all characters that are written after the '@' in the formatting string.

By default, the formatting string "<@>" is to be used here when working with the standard Arduino software of the BMSAIT. Adjustments are only necessary for special user requirements or when communicating with other Arduino programming.

#### Test

The value entered here is used to be able to send data to connected Arduino even without a connection to a running Falcon BMS when the test mode is selected.

#### Position

The value entered here is sent to the Arduino with each data packet. The value controls which line of the Arduino's data container the transferred value is written to. This value is therefore of great importance. A program logic ensures that the item number can only be assigned once. The item number cannot be 99. Values above 100 are only necessary in special cases (Image Backup Radio).

#### Update Frequency

If the record in the SharedMem is changed, the current value is always immediately transferred to the Arduino. However, if there is no change, data variables are still transmitted to reduce errors, but with a reduced frequency. The value entered here controls this by specifying the period after which an unchanged value will be transferred again.

### 5.2.3. Commands (Windows app)

#### ID

Clear identification of the command. When an Arduino transmits this ID as a command, the key combination stored for this ID is triggered.

#### Type

Classifies whether the command (1) triggers a button press or (2) joystick button or (3) affects a joystick axis.


#### Joystick

The vJoy software can simulate multiple joysticks. This is where the joystick is stored that is to be used to trigger a signal.

#### Key

For Keyboard Type:

A definition of a keystroke is stored in the format of a virtual keystroke of the VirtualInput Library.

 `enum` WindowsInputLib.Native.VirtualKeyCode  
The list of VirtualKeyCodes (see: [http://msdn.microsoft.com/en-us/library/ms645540\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms645540(VS.85).aspx))

For joystick button type:

The number of the joystick button is stored.

For joystick axle type:

```
Assembly vJoyInterfaceWrap,
{
    public enum HID_USAGES
    {
        HID_USAGE_X = 48,
        HID_USAGE_Y = 49,
        HID_USAGE_Z = 50,
        HID_USAGE_RX = 51,
        HID_USAGE_RY = 52,
        HID_USAGE_RZ = 53,
        HID_USAGE_SL0 = 54,
        HID_USAGE_SL1 = 55,
        HID_USAGE_WHL = 56,
        HID_USAGE_POV = 57
    }
}
```

The index of the selected joystick axis (0..9) is stored according to the vJoy HID\_Usages list.

#### Ctrl

Specifies whether the Control button is pressed together with the button.

#### Old

Specifies whether the Alt key is pressed together with the key.

#### Shift

Specifies whether the Shift key is pressed together with the key.

#### Mode

Determines whether a key is pressed, pressed, and released, or released.

### 5.2.4. Global Variables (Arduino)

#### BAUD RATE

The baud rate defined here must match the baud rate of the Windows app to enable communication. Default is 57600.

#### DATA LENGTH

This determines how many characters the Arduino has ready to record a data value. If the value is too high, it unnecessarily blocks the Arduino's scarce resources. If the value is too low, data cannot be completely transferred and passed on. The default value is 8 (7 usable characters plus one character for the word terminator required in the Arduino).

#### MESSAGEBEGIN

Here, a byte is defined that the Arduino can use to identify the beginning of a new message. It must therefore be a byte that never occurs in normal data exchange. By default, 255 is provided here.

#### HANDSHAKE

If the Arduino receives the byte defined here as InfoByte, a reply message will be sent back to the PC with the ID of this Arduino. Default is 128.

#### SWITCH POSITION

When the Arduino receives the InfoByte defined here, an input command is sent for all connected inputs (toggle switch, rotary switch) (see 2.2.3). Default is 150.

#### CALIBRATE

If the Arduino receives the byte defined here as InfoByte, motors will be recalibrated. This is necessary when using stepper motors in order to get them into a defined position before starting the program (see 2.2.1). Default is 160.

#### ZEROIZE

If the Arduino receives the byte defined here as InfoByte, motors will be reset to zero value. This is necessary when using stepper motors in order to get them into a defined position before starting the program (see 2.2.2). Default is 161.

#### STARTPULL

If the Arduino receives the byte defined here as InfoByte, the PULL processing will start (see 2.1.4.2). Default is 170.

#### FINAL PULL

If the Arduino receives the byte defined here as InfoByte, the PULL processing will stop (see 2.1.4.2). Default is 180.

#### TESTON

If the Arduino receives the byte defined here as InfoByte, the internal test mode is activated, which is used to report debug information back to the Windows app. Default is 190.

#### TESTOFF

If the Arduino receives the byte defined here as InfoByte, the internal test mode will end. Default is 200.

#### VAR\_BEGIN

Here, a character is defined that the Arduino can use to recognize that a data value begins immediately afterwards in a transmission. The default is '<'.

#### VAR\_ENDE

Here, a character is defined that allows the Arduino to recognize that the data value just read in is now complete. The default is '>'.

#### TYP\_ANFANG

This defines a character that allows the Arduino to recognize that the next character of a transmission is the specification of a data type. The default is '{'.

#### TYP\_ENDE

This defines a character that allows the Arduino to recognize that the previous character of a transmission is the specification of a data type. Default is '}'.

### 5.2.5. Internal commands

Using internal commands, Arduinos can use the Switches module (see 4.2.3) of the Windows app. These are hard-coded.

Byte Code	Description	Reference Documentation
253	Transferring Switch Positions	2.2.3
254	Bring engines to zero position	2.2.2
255	Engine calibration (full)	2.2.1



## 5.2.6. Font DED/PFL

C	A	Z	F		C	A	Z	F		C	A	Z	F		C	A	Z	F		C	A	Z	F	
1					48	0	0	0		69	E	E	E		90	Z	Z	Z		111	o		K	
2					49	1	1	1		70	F	F	F		91	[		0		112	p		L	
3-28																								
29					50	2	2	2		71	G	G	G		92	\		1		113	q		M	
30					51	3	3	3		72	H	H	H		93	]		2		114	r		N	
31					52	4	4	4		73	I	I	I		94	^		3		115	s		O	
32					53	5	5	5		74	J	J	J		95	_		4		116	t		P	
33	!				54	6	6	6		75	K	K	K		96	`		5		117	u		Q	
34	"				55	7	7	7		76	L	L	L		97	a		6		118	v		R	
35	#				56	8	8	8		77	M	M	M		98	b		7		119	w		S	
36	\$				57	9	9	9		78	N	N	N		99	c		8		120	x		T	
37	%				58	:	:	:		79	O	O	O		100	d		9		121	y		U	
38	&				59	;	;	;		80	P	P	P		101	e		A		122	z		V	
39	'				60	<	*	*		81	Q	Q	Q		102	f		B		123	{		W	
40	(				61	=	-	-		82	R	R	R		103	g		C		124			X	
41	)				62	>	>	>		83	S	S	S		104	h		D		125	}		Y	
42	*				63	?	°	°		84	T	T	T		105	i		E		126	~		Z	
43	+				64	@				85	U	U	U		106	j		F		127				
44	,				65	A	A	A		86	V	V	V		107	k		G		128				
45	-				66	B	B	B		87	W	W	W		108	l		H						
46	.				67	C	C	C		88	X	X	X		109	m		I						
47	/				68	D	D	D		89	Y	Y	Y		110	n		J						

C – Byte value A – ASCII character

Z – Code in the FalconBMS SharedMem F – Character FalconDED Font Arduino