Falcon BMS to Arduino Interface Tool (BMSAIT)

Falcon BMS to Arduino Interface Tool — EN

Author	**Robin "Hummer" Bruns**
English Translation	Francisco “Chisco” de Ascanio
Document Version	1.4
Software Version	1.8.4
BMS Version	4.37
Date	17.01.2023

Table of Contents

Contents

1. Introduction

1.1. Basics

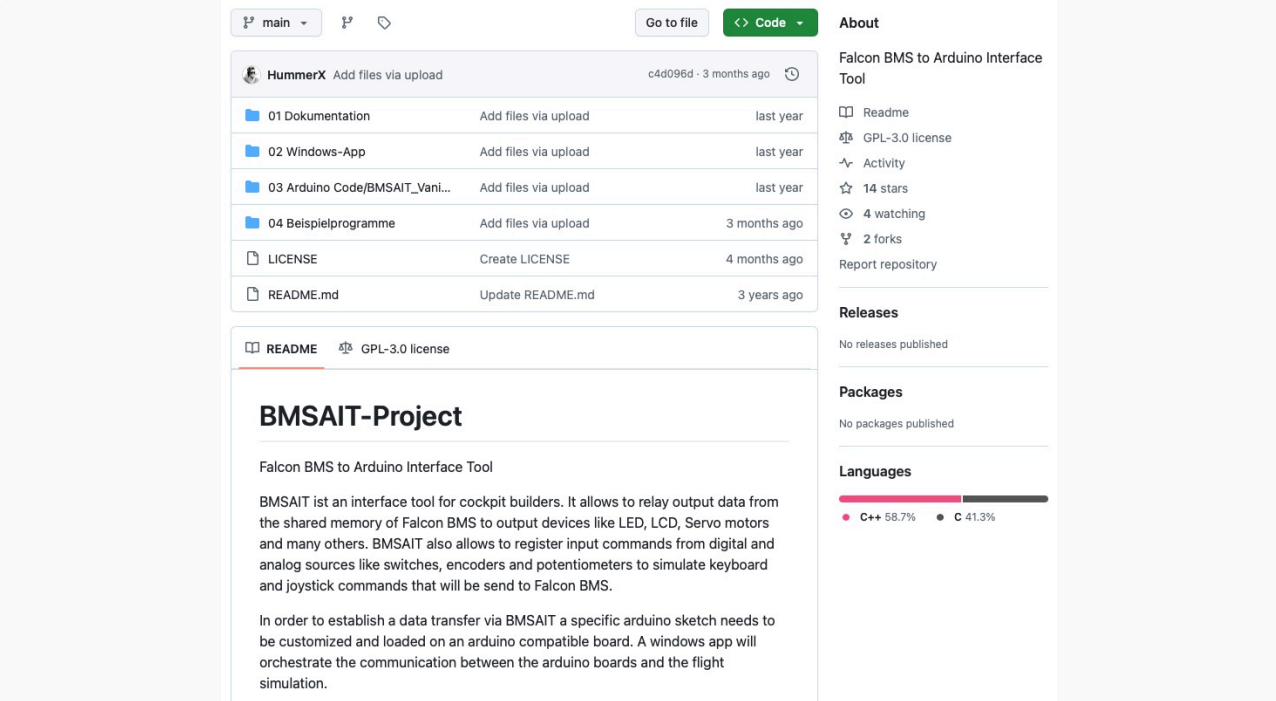
The Falcon BMS to Arduino Interface Tool (BMSAIT) is a tool that connects Falcon BMS flight simulation software to custom-built hardware for home cockpits. It allows simulation data to be displayed on devices like LEDs, motors, and screens controlled by Arduino boards, while also sending user inputs from switches or knobs back to the simulation.

It also works with DCS flight simulation software, allowing users to switch between Falcon BMS and DCS without reprogramming Arduino boards. The software automatically detects the active simulation and adjusts accordingly.

BMSAIT supports many Arduino models, such as the Uno, Mega, Nano, and ESP32, and works with a wide range of devices. These include basic components like LEDs and buttons, as well as advanced tools like stepper motors, seven-segment displays, and OLED screens for Falcon BMS's Data Entry Display (DED) and Pilot Fault List (PFL).

BMSAIT simplifies cockpit building by providing a flexible way to connect simulation software to physical controls and displays. It's a great tool for flight simulation enthusiasts creating custom cockpits.

The author of BMSAIT is **Robin "Hummer" Bruns**. The project files can be located at https://github.com/HummerX/BMSAIT.



The **Falcon BMS to Arduino Interface Tool (BMSAIT)** is a utility designed to:

1. Extract flight information from Falcon BMS and make it available for controlling devices in cockpit construction.
2. Process and send control signals to the flight simulation.

The tool consists of two components:

1. **The Windows Application**

   a. Extracts data from Falcon BMS's Shared Memory (SharedMem) and outputs the data via a serial interface.

   b. Receives control inputs from an Arduino and converts them into keyboard signals or joystick commands.

2. **The Arduino Sketch**

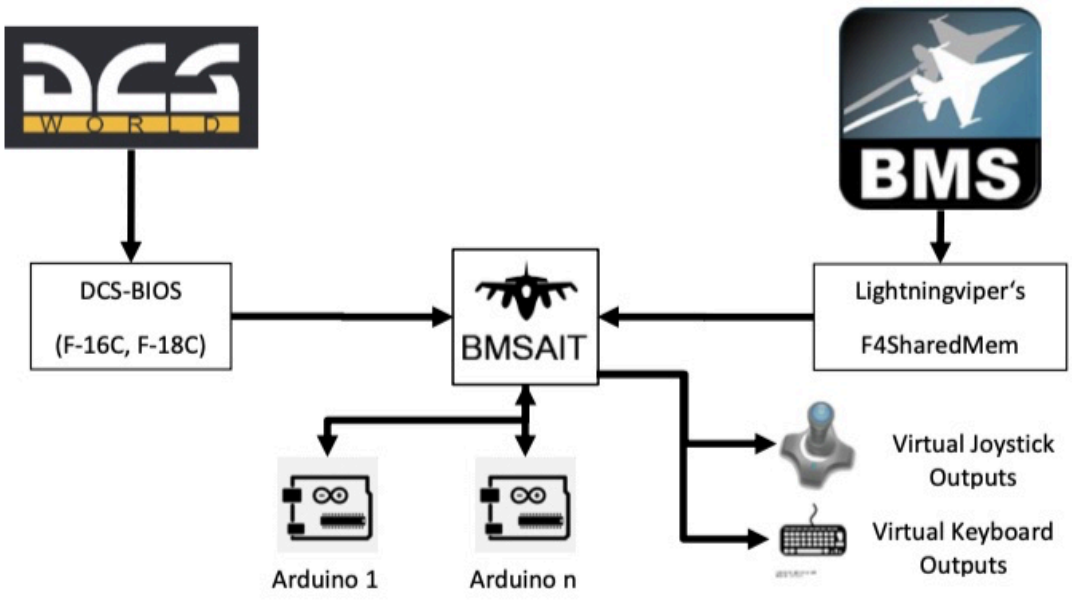
   a. Configures an Arduino board to receive data via the serial interface and output it to selected devices (e.g., LEDs, LCDs, servos, and others).

   b. Reads various input devices and sends control signals back to the Windows application.

In the past, many tools have been developed for extracting data from the Falcon BMS SharedMem (to name a few: FAST, BMSCockpit, DEDuino, F4toSerial).

In many areas, BMSAIT cannot compete with these programs, nor is it intended to. BMSAIT was developed out of necessity because many of these tools are either no longer maintained (and therefore do not provide access to all the data available in Falcon BMS's SharedMem after recent updates) or have a limited range of functionality, meaning they cannot support all cockpit projects.

At its current stage of development, BMSAIT should be viewed as a *complementary tool* for use cases that are not covered by existing programs.



The initial purpose of BMSAIT was to serve solely as an output program, with the generation of control signals via input devices being only a peripheral function. However, as the input functions were further developed, significant potential emerged. With BMSAIT software, an Arduino is now theoretically capable of representing all the functions of a cockpit panel in cockpit construction.

Connecting cockpit panels to central controllers can lead to complex and potentially error-prone structures (I speak from experience!). BMSAIT allows one or more panels to be connected to a single Arduino located directly behind the panel, which manages all the panel's inputs and outputs. As a result, the panel requires only a single USB cable (and, if necessary, an external power supply) to operate. This helps make cockpit construction more organized and manageable.

It is generally intended that the Windows tool is used in combination with the Arduino sketch. However, it is theoretically possible to use both tools independently in combination with other software.

If you want to use BMSAIT, you will need to deal with some hardware-related questions and a bit of Arduino programming in C++. I hope this documentation provides all the necessary explanations to enable you to set up the software even without programming knowledge. If in doubt, feel free to contact me—within the limits of my available free time, I will be happy to assist you with the setup or with your ideas for extensions.

BMSAIT can theoretically be used with all common Arduino boards. However, I have experienced communication issues 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, but the peculiarities of this type may lead to issues.

For some use cases, it may be beneficial for the microcontroller to have more processing power and memory (e.g., for controlling the DED). In such cases, it is possible to use BMSAIT on controllers such as the ESP32.

1.2 Quickstart

Start by reading Chapter 3.1 (First Steps for the Windows Application) and Chapter 4.1 (Arduino Preparation) to establish the prerequisites for using BMSAIT.

For initial experiments with BMSAIT, I recommend trying out the available examples. The example programs include:

- A preconfigured sketch for the Arduino (`*.ino` and the associated `*.h` files).
- A configuration file (`*.ini`) for the Windows application.
- A separate document with a detailed description of the required wiring (pdf).

Once the functionality of the software has been verified on your computer and you have understood the basics of the software, it will be easier for you to implement your own projects.

Key Features:

• Initially designed as an output program, BMSAIT now also supports input devices, enabling comprehensive cockpit panel functionality.

• Instead of wiring cockpit panels to central controllers (which can lead to complex and error-prone setups), BMSAIT allows each panel to be connected to its own Arduino. This requires only one USB cable per panel, simplifying the cockpit setup.

The Windows tool and Arduino sketch are primarily intended to work together but can also be used independently with other software.

Supported Arduino Boards

BMSAIT is compatible with most common Arduino boards. However, communication issues have been observed with the Leonardo/Pro Micro boards, so Uno, Micro, Nano, or Mega are recommended. Support for the Due board is available from version 1.2, but its unique characteristics may lead to problems.

For advanced use cases, microcontrollers with more power and memory (e.g., ESP32) can also be used.

User Skills Required

To use BMSAIT, basic hardware knowledge and some understanding of Arduino programming in C++ are required. This documentation aims to provide all necessary explanations for setup, even for those without programming experience. Assistance is available based on time availability.

For testing, start with the provided example programs. These include:

• Pre-configured Arduino sketches (*.ino and associated .h files).

• Configuration files (*.ini) for the Windows application.

• A detailed wiring guide.

Once the software functions as intended on your system and you understand the basics, implementing your own projects will become easier.

Example programs are available at:

https://github.com/HummerX/BMSAIT/tree/main/04%20Beispielprogramme

2. Core Concepts

2.1. Data Exchange Between PC and Arduino Board

2.1.1. Hardware Connection

Data exchange between Windows and Arduino is handled via a serial data connection, usually established through the USB connection of the Arduino. The Arduino is assigned a COM port. If connected to a different USB port, the Arduino may be assigned a different COM port number.

The Arduino is powered through the USB port. If the Arduino drives devices with high power requirements (e.g., servomotors), an additional power supply is recommended to prevent damage.

Note: When using an external power source for devices connected to the Arduino, ensure the ground GND connections of the Arduino and the external source are linked.

2.1.2. Software Connection

To enable communication between Windows and Arduino, specific software settings must be configured, including the BAUD rate (transmission speed). The BAUD rate must match between the Arduino and the Windows tool. If you want to change the BAUD rate, you need to modify the Arduino sketch and re-upload it (see Chapter 4.2.2). In the Windows tool, the BAUD rate can be set in the device settings before establishing a connection (see Chapter 3.2.3). Communication is only possible if both BAUD rates match.

BMSAIT supports two different communication methods for exchanging information with properly programmed Arduino boards:

• PUSH principle: The Windows application leads the data exchange.

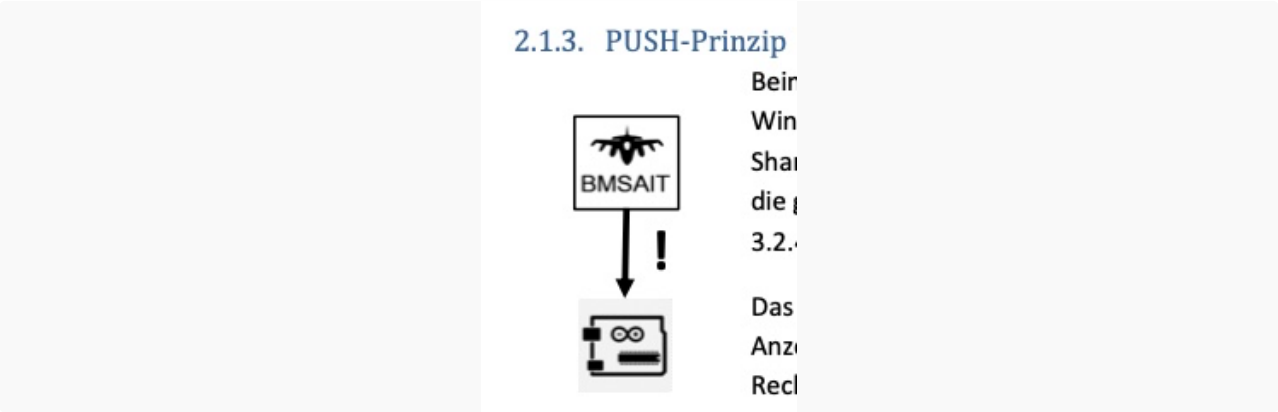
• PULL principle: The Arduino requests data as needed.

From version 1.3 onward, BMSAIT ***allows configuring either method for each Arduino individually***.

> Simple Difference Between Push and Pull Principle:
>
> 1. Push Principle:
>
> • Control: The PC (Windows application) is in control.
>
> • Data Flow: The PC sends data to the Arduino without being requested.
>
> • Configuration: Variables and data are pre-defined in the PC application.
>
> • Use Case: Suitable when the PC handles most of the logic, and the Arduino acts mainly as an output device (e.g., displaying or acting on incoming data).
>
> 2. Pull Principle:
>
> • Control: The Arduino is in control.
>
> • Data Flow: The Arduino requests specific data from the PC when needed.
>
> • Configuration: Variables are defined on the Arduino side.
>
> • Use Case: Ideal for ensuring synchronization and when time-critical data needs to be processed promptly by the Arduino.

> In essence:
>
> • Push: PC decides what and when to send.
>
> • Pull: Arduino requests what it needs when it needs it.

2.1.3. PUSH Principle



In the PUSH principle, the Windows application drives the data exchange. It selects relevant information from Falcon BMS's Shared Memory (SharedMem). An editor in the application allows users to configure which data is transmitted (see 3.2.4).

The Arduino board simply receives the data and outputs it to connected devices. This approach reduces the Arduino's processing load and simplifies function testing.

2.1.3.1. Workflow of the Windows Software in PUSH Mode

Processing Start:

Processing begins when the "Start Connection" button is clicked in the Windows application.

If the **Autostart** option has been enabled in the basic settings, processing begins immediately when the Windows program (`BMSAIT.exe`) is started.

Initialization (one-time):

1. Establish quick access to selected SharedMem variables.
2. Open the COM port for communication with the Arduino.

Loop (repeated indefinitely until the user stops the connection):

The frequency of the loop cycles can be adjusted via the `polltime` value in the basic settings (see Chapter 4.2.2).

1. Load the current snapshot of Falcon BMS's SharedMem.
2. Output data to the Data Monitor.
3. For each configured data variable:
   - Retrieve the desired value from SharedMem based on its data type and ID.
   - Convert the value to ASCII format.

- Prepare the data packet for transmission, including control information.
- Send the data packet via the serial port.

4. Check for responses from the Arduino on any COM port.
5. Display messages in the text console of the main form.
6. Process keyboard command instructions and trigger key signals.

Processing End:

Processing ends when the user clicks the "End Connection" button. Naturally, closing the `BMSAIT` program also ends the processing.

2.1.3.2. Workflow of the Arduino Software in PUSH Mode

Processing Start:

The Arduino begins processing as soon as it is powered on and a valid program is loaded.

Initialization (at each connection setup):

1. Predefine general variables.
2. Initialize software for controlling peripherals (import libraries and assign variables).
3. Open the COM port for communication.

Loop (repeated indefinitely):

1. Monitor the state of connected switches and analog axes. Send commands to the PC if changes are detected.
2. Listen for system commands from the PC and process them.
3. Listen for simulation data from the PC. If data is received:
   - Read and temporarily store it.
4. Update displays connected to the Arduino.

Processing End:

Processing ends when the Arduino is disconnected from power or its current program is overwritten.

2.1.3.3. Data Format for PC-to-Arduino Transmission in PUSH Mode

Principle

`<START-Byte><INFO-Byte><FormatString1><Data><FormatString2>`

> A transmission consists of the following components:
>
> - START-Byte: Byte 255 signals the start of a transmission (see 4.2.2).
> - INFO-Byte: Indicates the type of message (e.g., handshake, data transmission).
> - FormatString1: Specifies characters preceding the data. Default is < (see 4.2.2 “VAR_BEGIN”).

> • Data: The actual value extracted from SharedMem. This could be a single character (e.g., a boolean) or a string of multiple characters (e.g., a line from the DED). The maximum data length is defined in the Arduino sketch.
>
> • FormatString2: Specifies characters following the data. Default is > (see 4.2.2 "VAR_END").

START-Byte:

The byte 255 signals the start of a transmission (see Chapter 4.2.2).

INFO-Byte:

The INFO-Byte indicates to the Arduino what type of message is being transmitted:

- **Handshake:**

  If a defined byte is transmitted (see Chapter 4.2.2 "HANDSHAKE"), it signals to the Arduino that the Windows application is currently searching for connected Arduino boards. The Arduino will respond with a defined reply signal (the board's ID, see Chapter 4.2.2.2) to notify the Windows application that the message has been received and the board is ready.

- **Control PULL:**

  See Chapter 2.1.4.4.

- **Data Transmission:**

  During a data transmission, the INFO-Byte specifies the position number in which the value of the current transmission should be temporarily stored in the Arduino's prepared data container.

FormatString1:

In the Windows application, formatting options can be specified during variable mapping. FormatString1 contains the characters that should appear before the actual data value. By default, the BMSAIT Arduino sketch expects the character `<` here (see Chapter 4.2.2 "VAR_BEGIN").

Data:

This refers to the information read from the SharedMem. It can be:

- A single character (e.g., a true/false value).
- A string containing many characters (e.g., a line from the DED).

The maximum length for a data record can be defined in the Arduino sketch. This value should range between:

- **1:** Exclusively for processing LEDs with true/false values.
- **25:** For a full line of the DED or PFL.

FormatString2:

In the Windows application, formatting options can be specified during variable mapping. FormatString2 contains the characters that should appear after the actual data value. The provided Arduino sketch expects the character `>` here (see Chapter 4.2.2 "VAR_END").

2.1.3.4. Data Format for Arduino-to-PC Transmission in PUSH Mode

A transmission consists of the specification of the transmission type and the actual data record:

```
<Transmission Type><Data Record>
```

Transmission Type	Content	Data Record
t	The transmitted data record is to be displayed in the console of BMSAIT.	
k	The current status of a switch is transmitted (1 = active, 0 = not active).	
a	The transmitted data record contains the position information of an analog axis:	`<Command ID>,<Axis Position>`
	The Command ID is a three-digit number that refers to an entry in the command management of the Windows application.	
	If a command with this ID is assigned to the Arduino in the Windows application and linked to an analog axis of vJoy, the vJoy analog axis is set to the position (0..1024) transmitted by the Arduino.	
g	The Arduino reports that the input buffer is empty and ready to receive new data.	
s	The Arduino sends an internal command to the BMSAIT app.	

A transmission consists of:

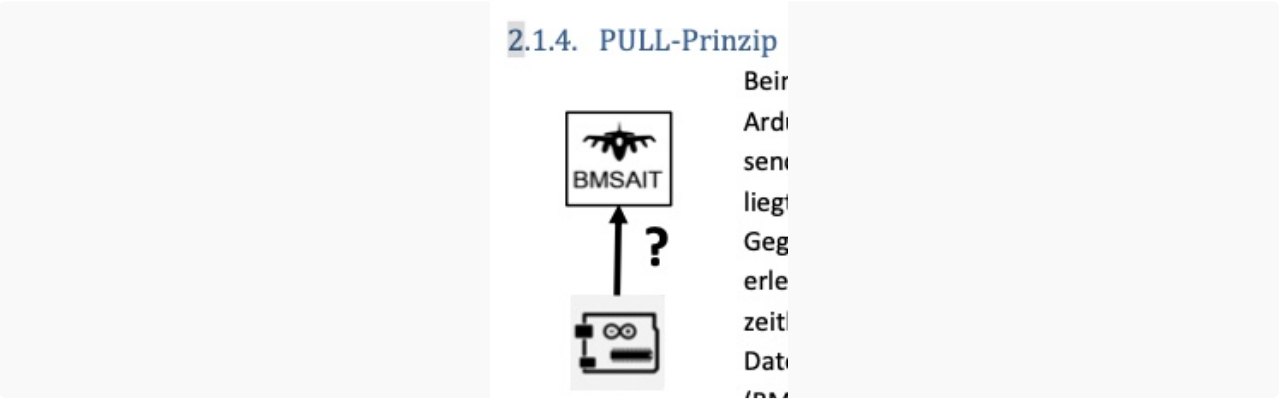
Transmission Types:

- t: Data set should be displayed in the BMSAIT console.
- k: Current switch status (1 = active, 0 = inactive).
- a: Analog axis position, formatted as , (e.g., 1024). `<CommandID>, <AxisPosition>`

  The command ID is a three-digit number that links to the Windows application. When assigned to a virtual joystick, the joystick's axis position will correspond to the value sent by the Arduino.

  If a command with this ID is assigned to the Arduino in the Windows application and linked to an analog axis of vJoy, the vJoy analog axis is set to the position (0..1024) transmitted by the Arduino.
- g: Signals that the Arduino's input buffer is empty and ready for new data.
- s: Sends an internal command to the BMSAIT application.

2.1.4. PULL Principle



In the PULL principle, the Arduino software drives the data exchange. The Windows software waits for requests from the Arduino and only sends data when explicitly asked. This simplifies configuration since desired variables only need to be set up in the Arduino sketch. PULL improves synchronization, ensuring time-sensitive data is transmitted more promptly.

The advantage of this approach is that *the configuration of the desired data variables only needs to be done once* (namely in the Arduino) as opposed to PUSH mode. PULL also facilitates temporal synchronization of data exchange, ensuring that time-critical data reaches the Arduino faster. All required data variables must—just as in PUSH mode—be entered in the Arduino sketch (`BMSAIT-UserConfig.h`).

2.1.4.1. Workflow of the Windows Software in PULL Mode

The PULL processing mode can be selected for an Arduino device in the Windows application's settings.

Processing Start:

When the “Start Connection” button is clicked, the application sends a code signal to the Arduino to activate PULL logic.

Initialization (one-time):

1. Send a code signal to activate the PULL logic on the Arduino.
2. Establish a connection to Falcon BMS or DCS SharedMem.

Loop (repeated indefinitely):

1. Read current data from SharedMem.
2. Display SharedMem data on the data monitor.
3. Listen for Arduino requests or commands:
   - Read requested values from SharedMem.
   - Convert them to ASCII format.
   - Prepare the data packet and send it via the serial port.
4. Process key/joystick signals and update analog axes.

Processing End:

When the “End Connection” button is clicked, the application signals the Arduino to stop sending PULL requests.

2.1.4.2. Workflow of the Arduino Software in PULL Mode

Start of Processing:

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

Initialization:

- Predefine variables.
- Start the software for controlling peripherals.
- Open the COM port.

Loop:

The loop runs continuously on the Arduino until processing ends.

- Listen for commands from the PC:
  - If commands are present, process them.
  - If the command to activate the PULL principle is received, the following PULL processing steps are executed.
  - If the command to deactivate the PULL principle is received, PULL processing is no longer executed.
- **PULL Processing (only if activated):**
  - Iterate through all data variables registered in the data container.
  - Prepare a command to request data for the data variable.
  - Send the data request.
- Listen for serial data from the PC:
  - If data is present, read and temporarily store it.
- Output the temporarily stored data via connected peripherals.
- Check the state of connected switches. If a change is detected, send commands to the PC.

End of Processing:

Processing can only be terminated by disconnecting the Arduino from the power supply

2.1.4.3. Data Format for a Transmission from Arduino to PC in the PULL Principle

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

- **d:**

  The transmitted data record contains a data request in the format:

  `<DataPosition>,<VariableType>,<VariableID>`

  - **DataPosition:**

    This is the position in the data container of the Arduino sketch.

  - **VariableType:**

This is a single byte indicating the type of the data variable, e.g., **{f}** for a float number. The value is read from the data container (`Datenfeld.format`).

If an incorrect format is specified in the data container, BMSAIT will not process the request. Refer to the description of data variables in the appendix for more details.

- **VariableID:**

  This is the ID of the data variable, taken from the data container (`Datenfeld.id`). For further processing, it is essential that this is a four-character text.

  When filling the data container, ensure that leading zeros are included!

- **u:**

  With this command, the BMSAIT app is asked to transmit the current data of the DED ("uDED") or PFL ("uPFL").

  The Windows application will then prepare the five lines of the DED/PFL and send them sequentially to the requesting Arduino.

d	Der übertragene Datensatz enthält eine Datenanfrage im Format: <Datenposition>,<Variablentyp>,<VariablenID> **Datenposition** Dies ist die Position im Datencontainer des Arduino-Sketch **Variablentyp** Hier wird mit einem Byte der Variablentyp der Datenvariablen angezeigt, z.B. {f} für eine float-Zahl. Der Wert wird aus dem Datencontainer gelesen (Datenfeld.format). Wenn im Datencontainer ein falsches Format angegeben ist, wird die Anfrage durch BMSAIT nicht bearbeitet werden können. Siehe auch die Beschreibung der *Datenvariable* in der Anlage **VariablenID** Hier wird aus dem Datencontainer die ID der Datenvariablen übernommen (Datenfeld.id). Für die weitere Verarbeitung ist wichtig, dass es sich um einen Text mit genau vier Zeichen handelt. In der Befüllung des Datencontainers ist daher darauf zu achten, dass führende Nullen mitgegeben werden!
u	Mit diesem Kommando wird die BMSAIT App um Übertragung der aktuellen Daten des DED („uDED“) oder PFL („uPFL“) gebeten. Die WinApp wird daraufhin die fünf Zeilen des DED/PFL aufbereiten und nacheinander an den anfordernden Arduino senden.

2.1.4.4. Data Format for a Transmission from PC to Arduino in the PULL Principle

The format follows Chapter 2.1.3.3, with the addition that two INFO bytes are defined to control the PULL processing on the Arduino:

- **Start PULL Processing:**

  When the INFO byte value **0xAA (170)** is transmitted, it signals to the Arduino to start PULL processing (see Chapter 5.2.4 "STARTPULL").

- **End PULL Processing:**

  When the INFO byte value **0xB4 (180)** is transmitted, it signals to the Arduino to stop PULL processing (see Chapter 5.2.4 "ENDPULL").

2.2. Synchronization

2.2.1. Motor Calibration

For stepper motors, the software typically cannot retain the current position of the indicator. After restarting the Arduino, the software does not know the position of a pointer before the restart.

For this reason, stepper motors must be calibrated before being used in the simulation to align the software and hardware positions.

Motor calibration involves moving a motor to its zero position (full counterclockwise rotation), performing the full range of motion clockwise, and then returning to the zero position. This ensures that motors are always correctly aligned to their zero position.

1. Move the motor to its zero position (full counterclockwise rotation).
2. Perform the full range of motion clockwise.
3. Return the motor to the zero position.

Calibration Methods:

1. **Via the BMSAIT Windows App Main Form:**
    - The "Calibrate Motors" button can be pressed, sending a signal to all configured Arduinos to calibrate the connected motors.
2. **Device-Specific Calibration:**
    - Using the selection menu for configured devices, a specific calibration can be instructed for a particular Arduino.
3. **Via a Button on the Arduino:**
    - A button configured on an Arduino using BMSAIT can be pressed to send an internal command (command ID 255) to the BMSAIT Windows App, triggering motor calibration.

2.2.2. Bringing Motors to Zero Position

A full motor calibration is not always appropriate, as it can take too long. Therefore, a "small" calibration option is available, where the motors are simply moved to their zero position.

This can be achieved in the following ways:

1. By pressing the "Calibrate Motors" button on the main form of the BMSAIT Windows App.
2. An automated calibration is triggered when exiting the 3D world of BMS or switching to the 2D world of DCS.
3. As a final option, a button connected to an Arduino can be used to send a synchronization request via an internal command to the BMSAIT Windows App (internal command 254).

2.2.3. Synchronizing Switch Positions

When starting the simulation, the switch positions in the simulation may not match those in the home cockpit. A switch position in the simulation is only updated when a physical switch is toggled, establishing synchronization.

To achieve quick synchronization, *it is possible to instruct the Arduinos with BMSAIT to transmit the positions of all switches*. This updates the simulation to reflect the positions of the physical switches.

There are three ways to achieve this (which, for obvious reasons, are only useful when in the 3D world of the simulation):

1. By pressing the "Calibrate Motors" button on the main form of the BMSAIT Windows App.
2. Synchronization is automatically triggered upon entering the 3D world of BMS.
3. As a final option, a button connected to an Arduino can be used to send a synchronization request via an internal command to the BMSAIT Windows App (internal command 253).

3. Windows Application

The Windows application was developed using Windows Visual Studio 2019 as a Windows app in C# for x64 processors. It utilizes .NET components version 4.7.2.

The following libraries are used for operation:

- **`F4SharedMem.dll`** by LightningViper for accessing the Shared Memory of Falcon BMS.
- **`WindowsInputLib.dll`** (Input Simulator Plus) by Michael Noonan and Theodoros Chatzigiannakis for generating keyboard signals.
- **`vJoyInterface.dll`** and **`vJoyInterfaceWrap.dll`** for generating joystick commands.

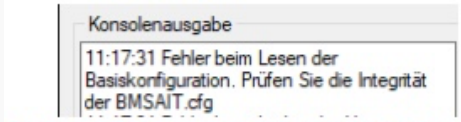
The project files with my source code are available upon request.

3.1. First Steps

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

- `BMSAIT.exe`
- `F4SharedMem.dll`
- `WindowsInput.dll`
- `WindowsInputLib.dll`
- `vJoyInterface.dll`
- `vJoyInterfaceWrap.dll`
- `BMSAITVariablen.csv`
- `BMSAIT-GaugeTable.csv`
- Subfolder **`de`** containing the file `BMSAIT.resources.dll` for German language settings.
- Subfolder **`en`** containing the file `BMSAIT.resources.dll` for English language settings.

When starting the program for the first time, an error message will appear because the basic settings must first be configured.



bbildung 1: Fehlermeldung bei fehlender Basiskonfiguration

![Figure 1: Error message when basic configuration is missing]

Updating the basic settings is described in Chapter 3.2.2.

3.2. BMSAIT Windows Software

3.2.1. The Main Form

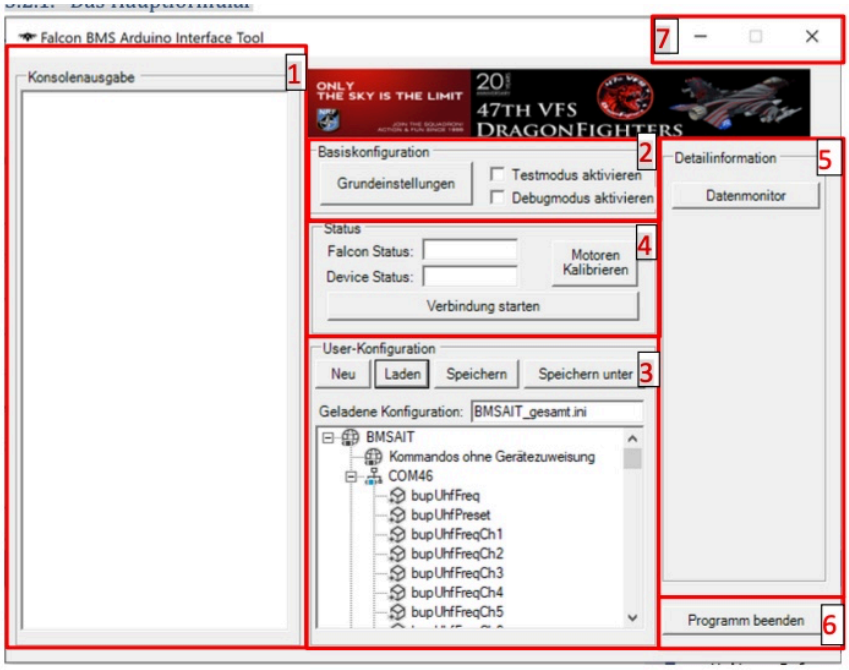


Abbildung 2: BMSAIT Hauptformular

![Figure 2: BMSAIT Main Form]

When the program starts, the main form is displayed. The main form provides information about the current settings and allows control of the application.

The main form is divided into several sections:

Main Form Section 1 – Console Output

This field displays status messages and error messages.

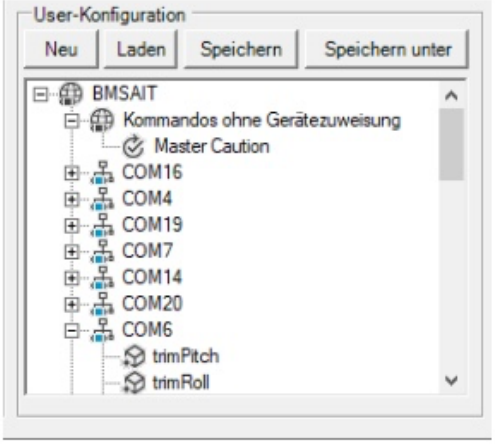
Main Form Section 2 – Basic Configuration

The **"Basic Settings"** button opens a form where the program's basic settings can be adjusted.

The **Test Mode** flag allows testing of the data connection between the Windows program and connected Arduinos without needing to start Falcon BMS. When this flag is set, and processing is started, no connection to Falcon BMS's Shared Memory will be established. Instead, test data entered in the configuration when defining a data variable will be used. In Test Mode, input commands reported by the Arduinos will also be triggered, even if Falcon BMS or DCS is not running.

In **Debug Mode**, status and error messages from the Arduino are displayed in the console for analysis.

Main Form Section 3 – User Configuration



This section provides tools for creating a configuration for devices (COM ports representing Arduinos), Shared Memory data variables (variables), and input commands.

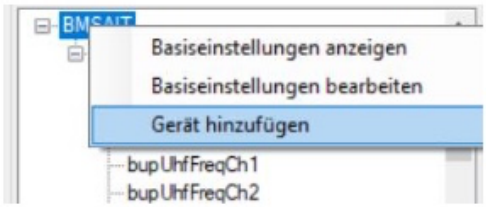
The window displays three levels:

- **Level 1** is the **TOP Node**.
- Below the TOP Node, devices (COM ports representing an Arduino) are created.
- At the lowest level, variables and input commands assigned to these devices are displayed.

The node **"Commands Without Device Assignment"** displays input commands that are not specifically assigned to an Arduino. Since input commands have only been directly assignable to an Arduino starting with version 1.3, when loading older configurations, input commands are displayed under this node. For new configurations, this node is irrelevant.

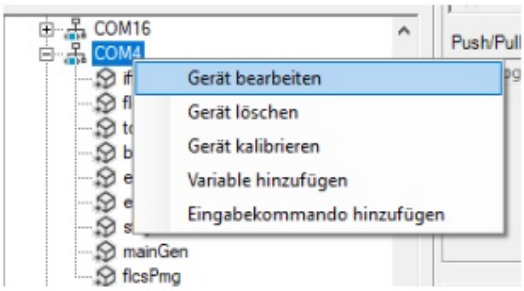
A left mouse click on an entry in the User Configuration window selects it. Detailed information about the selected object is displayed in **Section 5** of the main form.

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



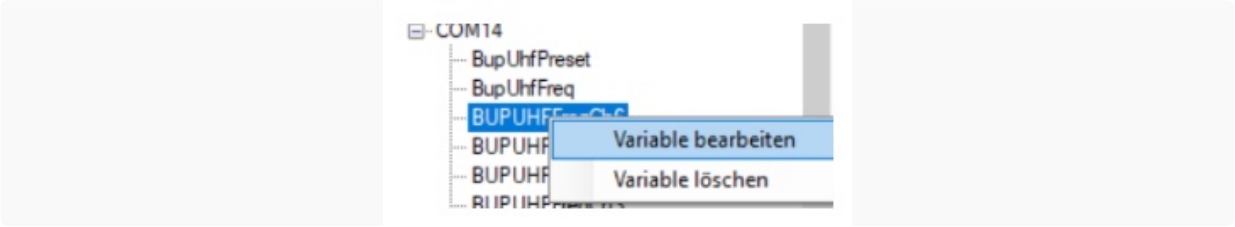
- **Show/Edit Basic Settings:**
  Opens a form where the program's basic settings can be adjusted.
- **Add Device:**
  Opens a window for adding an Arduino.

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



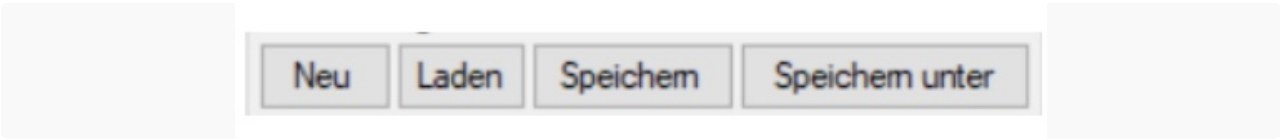
- **Edit Device:**
  Opens a window displaying the current settings of the selected device, which can then be modified.
- **Delete Device:**
  Deletes the selected device, including all associated variable assignments.
- **Calibrate Device:**
  Recalibrates the motors on the selected device.
- **Add Variable:**
  Opens a window for adding a variable.
- **Add Input Command:**
  Opens a window for adding input commands.

By **right-clicking on a variable**, the following options are available (only in PUSH mode):



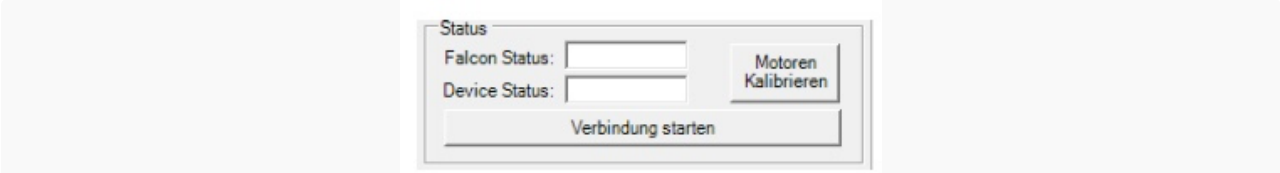
- **Edit Variable:**
  Opens a window where the current settings of the selected variable can be viewed and modified.
- **Delete Variable:**
  Deletes the assignment of the variable to the device.

In **Section 3**, the following buttons are also available:



- **New:**
  Deletes the current user configuration and replaces it with an empty configuration.
- **Load:**
  Opens a file selection dialog to choose a user configuration. If a valid configuration is selected, the corresponding data is updated in this section.
- **Save:**
  Saves the current user configuration to the default storage location. (The default storage location is specified in the basic settings.)
- **Save As:**
  Saves the current user configuration to a new file. A file selection dialog is opened, allowing the desired storage location and file name to be chosen.

Main Form Section 4 – Processing Control

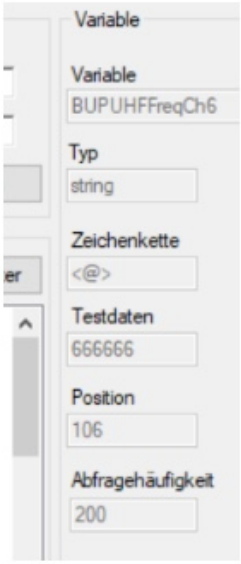


This section is used to start the program's main processing logic.

- The **"Falcon Status"** field displays the connection status with Falcon BMS or DCS. The status distinguishes between no connection, connection (2D world), and connection (3D world).

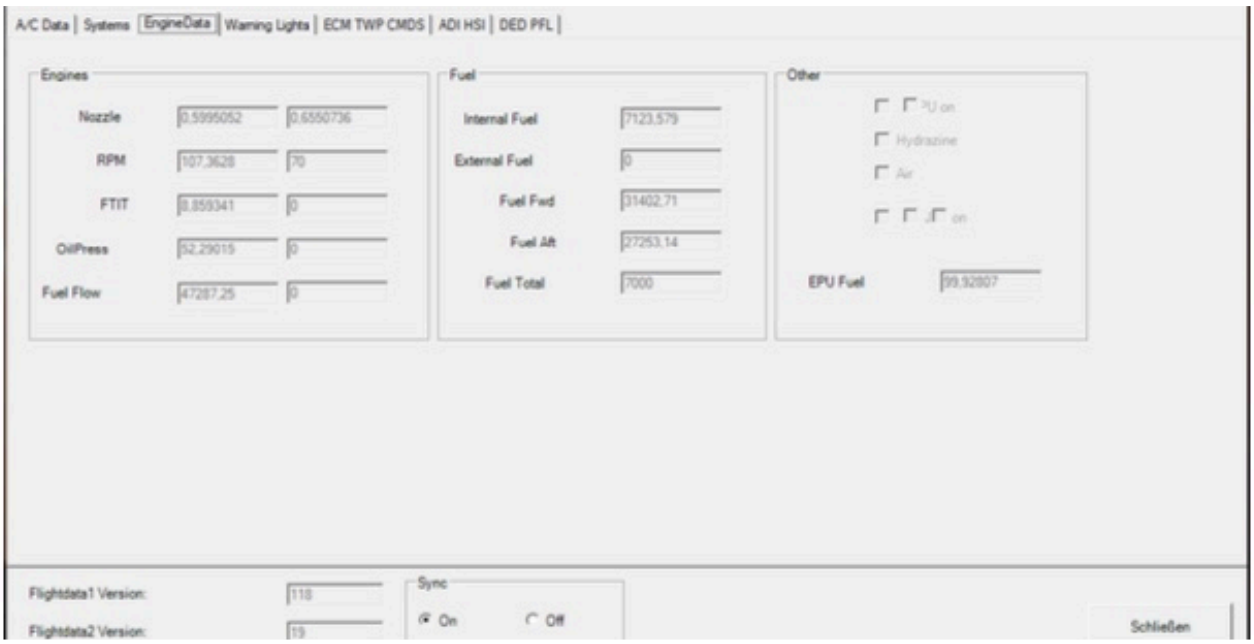
- The **"Device Status"** field indicates whether a COM connection to the Arduino(s) can be established during processing:
  - **Green:** Connection established with all defined Arduinos.
  - **Red:** No connection established.
  - **Yellow:** Connection established with some, but not all, defined Arduinos.
- The **"Calibrate Motors"** button temporarily establishes a connection with all configured Arduinos and sends the command to calibrate all connected devices. This is particularly relevant for stepper motors to bring their positions to the zero point. See Chapter 2.2 for more details.
- The **"Start Processing"** button initiates the program's main loop. The main loop continuously reads the current data from the SharedMem of Falcon BMS, manages data transmission to the Arduino(s), and triggers signals via a joystick/keyboard controller.

Main Form Section 5 – Detailed Information



This section displays detailed information depending on what is selected in the configuration window (Section 3). It could show details about the TOP Node, a device, a variable, or an input command.

By clicking the **"Data Monitor"** button, the data monitor can be started.



The **Data Monitor** includes almost all data fields available in the SharedMem. For clarity, these are divided into different tabs.

The Data Monitor has two functions, controlled via the synchronization mode:

- **Sync = On:**

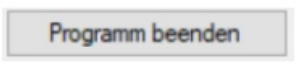
  The Data Monitor is used to display data read from the SharedMem of BMS/DCS. It can be used for troubleshooting when displays do not behave as expected. In synchronized mode, data fields cannot be modified.

- **Sync = Off:**

  The Data Monitor serves as a console for entering data. Data fields can be edited, and changes are transmitted to the Arduinos. This can be used to test functions such as LEDs, motors, etc.

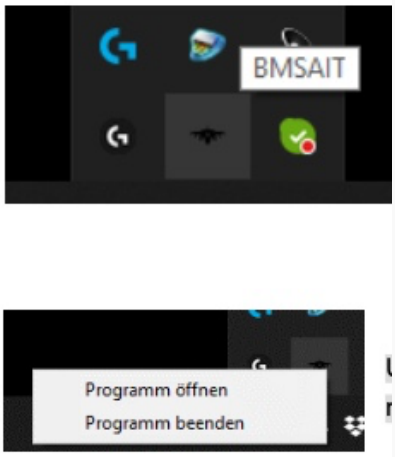
Main Form Section 6 – Exit Program

This section is used to close the program.



Main Form Section 7 – Window Management

This section manages the program's windows.



Here, the program can be closed by clicking the **X** button.

Clicking on the **Minimize** button reduces the program's display. Any ongoing processing will continue in the background.

The BMSAIT program will then be displayed as an icon in the taskbar.
To reopen the program as a window, right-click on the icon and select the command **"Open Program"**.

3.2.2. Basic Settings

The software creates a configuration file named `BMSAIT.cfg` in the same folder as the executable file. This file stores the selected basic settings and is automatically loaded when the program starts.

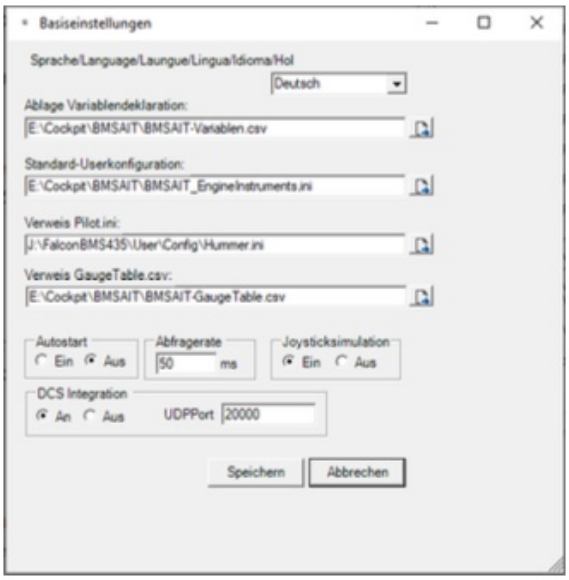


Abbildung 3: BMSAIT Basiseinstellungen

![Figure 3: BMSAIT Basic Settings]

The basic settings include:

- **Language Selection**

Here, the display language for BMSAIT can be switched (currently available in German or English).

- **Variable Declaration Directory**

  Use the selection tool to specify the folder where the `BMSAIT-Variablen.csv` file is located. This entry is mandatory.

- **Default User Configuration**

  This specifies which configuration (assignment of Arduino devices, variables, keyboard commands) should be automatically loaded when the program starts. This entry is optional.

- **Pilot.ini Reference**

  For displaying additional manual frequencies on the backup radio, specify the location of the `Pilot.ini` file in the `User/Config` folder of Falcon BMS. This entry is only necessary if backup frequencies are desired (see the BUPRadio example).

- **GaugeTable.csv Directory**

  Use the selection tool to specify the folder where the `BMSAIT-GaugeTable.csv` file is located. This entry is mandatory.

- **Autostart**

  This option determines whether the program immediately starts communication with Falcon BMS and the selected Arduinos upon launch. The default value is "Off."

- **Polling Rate**

  This setting specifies the interval between queries for current data from Falcon's SharedMem. The default value is 200ms. For precise instrument control (e.g., analog displays via stepper/servo motors), this value should be reduced.

- **Joystick Simulation**

  If activated, BMSAIT establishes connections with one or more virtual joysticks to send signals to Falcon BMS. This requires the "vJoy" software to be installed on the system and at least one virtual joystick configured (see Chapter 3.3).

- **DCS Integration**

  If enabled, BMSAIT can also process and transmit flight information from DCS. See the relevant chapter.

- **UDP Port**

  Specifies the UDP port for receiving flight information from DCS via DCSBIOS.

3.2.3. Configuring Devices

This form is used to set up a new device or edit an existing one.

In the **COM Port** field, select the port through which communication with the Arduino should occur. The dropdown list will display all COM ports known to Windows. If the COM port of an Arduino does not appear here, disconnect and reconnect the USB connection to the Arduino. After reopening this form, the COM port should be available.

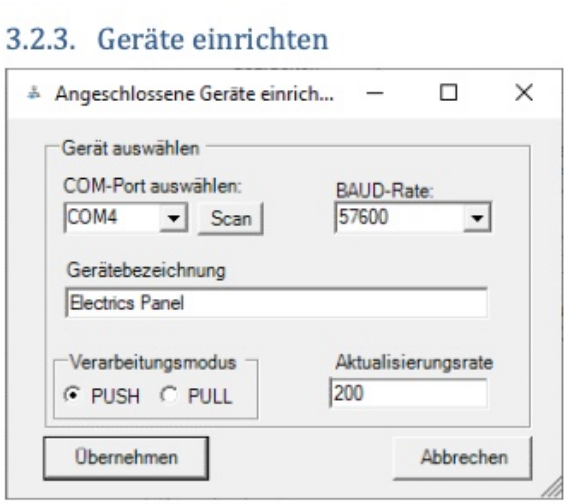


Abbildung 4: BMSAIT Geräteeinstellungen

![Figure 4: BMSAIT Device Settings]

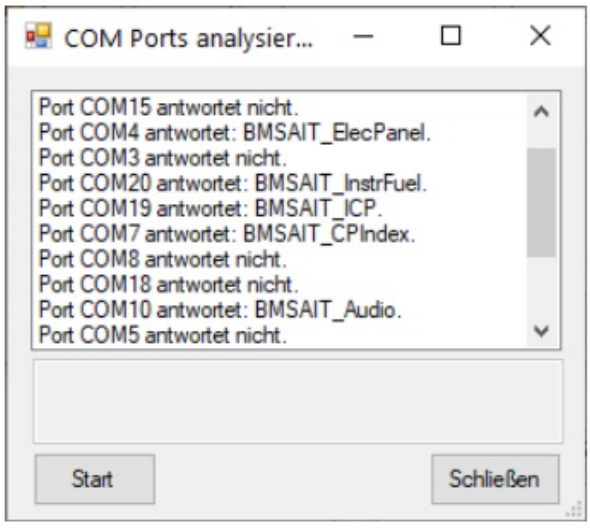
In the **BAUD Rate** field, specify the communication speed between the PC and the Arduino. The value entered here must match the value used in the Arduino's programming (see Chapter 4.2.2).

In the **Device Name** field, you can assign a name to the Arduino for easier identification.

The **Processing Method** determines the logic used for data exchange between the Windows app and the Arduino (see Chapters 2.1.3 and 2.1.4).

The **Refresh Rate** specifies how often data is sent to the Arduino for updates. The value should be between 50ms and 500ms, with 200ms being the default.

The **Apply** button saves the changes after performing a plausibility check on the inputs.



By clicking the **Scan** button, a subform is opened. Here, pressing the **Start** button checks all available COM ports on the PC and provides feedback about the status of the connections and any detected Arduino boards. If one of the BMSAIT sketches has been loaded onto an Arduino, the programmed ID of the Arduino (see Chapter 4.2.2 "ID") will be displayed here.

3.2.4. Variable Selection Form

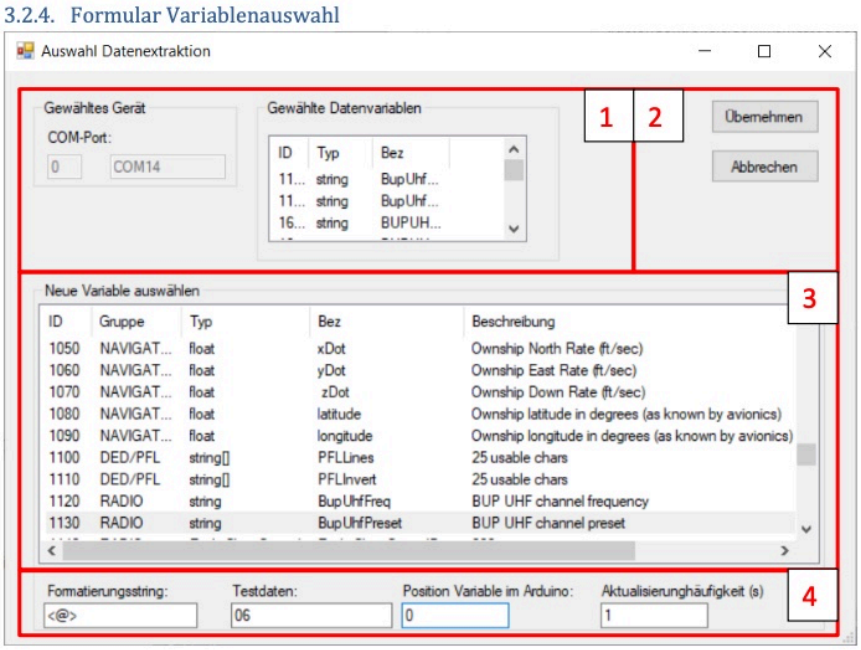


Abbildung 6: BMSAIT Variablenauswahl

![Figure 6: BMSAIT Variable Selection]
This form is accessed from the main form when, in the User Configuration, a right-click on a COM port selects the **"Add Variable"** command or a right-click on an existing variable selects the **"Edit Variable"** command.

The form for editing data variables is divided into the following sections:

Section 1 – Reference Data

In this section, the device/COM port currently selected on the main form is displayed. Additionally, a list of variables already assigned to the device is shown.

Section 2 – Completion

- The **Apply** button adds the inputs made to a new or existing data variable. Before saving, several plausibility checks are performed to prevent invalid entries. After saving is complete, the form is closed.
- The **Cancel** button closes the form without saving any changes.

Section 3 – Selecting a Data Variable

This window displays all available data variables in BMSAIT. These variables are loaded at program startup from the external file `BMSAIT-Variablen.csv`.

The sorting of the list can be adjusted by clicking on the column headers.
An element can be selected from the list. If you want to assign multiple data variables to a device, this must be done through multiple uses of this form.

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

Section 4 – Additional Features

Here, additional features for a data variable can be entered. Without filling out the **Position** field, saving is not possible. All other fields are optional, but it is recommended to populate them with meaningful values.

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

3.2.5. Input Commands Form

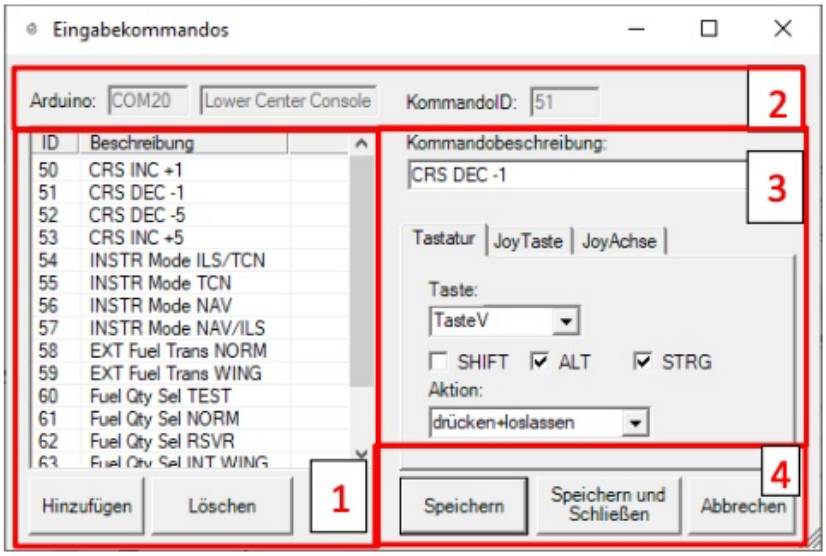


Abbildung 7: BMSAIT Eingabekommandos

![Figure 7: BMSAIT Input Commands]

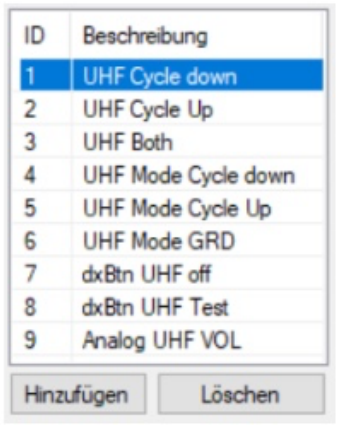
This form is accessed via the button in the Basic Settings section on the main form.

With the Input Commands form, keyboard or joystick signals can be defined. These signals are assigned an identification number (**Command ID**).

When a Command ID is sent by the Arduino, the corresponding signal is converted by the Windows application into a keyboard or joystick signal and sent to the currently active application.

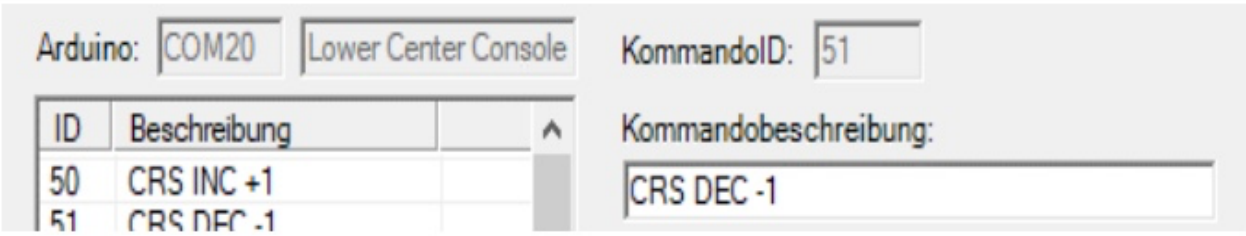
Note:

- BMSAIT sends a signal only when Falcon BMS or DCS is running (in 2D or 3D mode) or when the **test mode** is activated.



Section 1 – Selection List

- Displays all commands associated with the selected Arduino device in the current user configuration.
- **Functionality:**
  - You can select an existing entry by clicking on it. The related information is displayed on the right side of the form for editing.
  - **Add Button**: Creates a new command.
  - **Delete Button**: Removes the currently selected command.



Section 2 – Header Information

In the header information, the currently selected Arduino device for which the commands are to be managed is displayed. The **Command ID** field shows the identification number of the signal.

For a command that has not yet been saved, the value **NEW** is displayed until it is saved.

Below this is a field for a textual description of the command. This is intended to help identify the commands within BMSAIT and can be freely defined.

- Shows the currently selected Arduino device for which commands are being managed.
- **Command ID Field:**
  - Displays the signal's unique identification number.
  - If the command is new and not yet saved, the value **NEW** is displayed until saved.
- **Description Field:**
  - Allows for a textual description of the command for easier identification within BMSAIT.
  - The description can be freely defined.

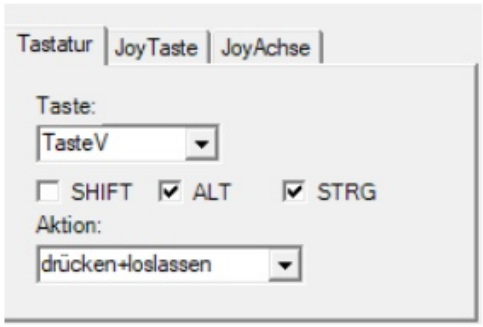
Section 3 – Mode Selection and Detailed Information

BMSAIT supports both triggering keyboard signals and using joystick commands or axes. In the mode selection, you can specify the type of signal the command should trigger.

If joystick processing is activated and the **vJoy software** is installed (see Chapter 3.3), three options are available. If **vJoy**is not activated, only the keyboard signal option is available.

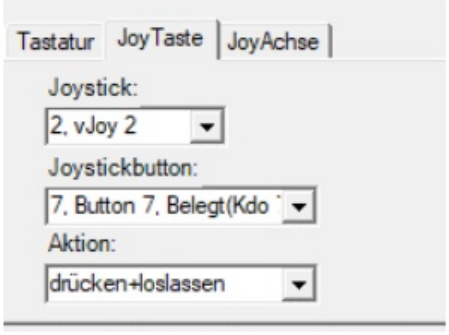
- **Modes Supported:**
  - **Keyboard Signals:** Standard keyboard input.
  - **Joystick Commands or Axes:** Requires vJoy software to be installed (see Chapter 3.3).
- If **Joystick Processing** is activated and vJoy is installed:
  - Three signal options are available.
- If **vJoy is not activated**, only the keyboard signal option will be accessible.

Keyboard Signals Tab



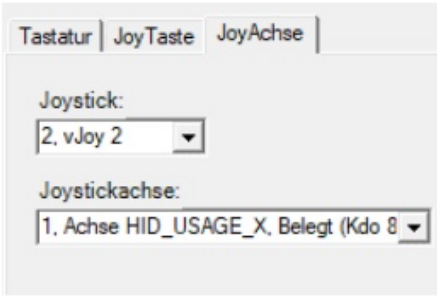
- In the **Key** field, a keystroke can be selected from a dropdown list.
- If the keystroke should be combined with a modifier (Shift, Alt, or Ctrl), the corresponding checkbox must be ticked.
- The **Action** setting defines the key behavior:
  - **Press only**
  - **Release only**
  - **Press & Release**

Joystick Button Tab



- First, select the desired joystick from the dropdown in the **Joystick** field.
- In the **Joystick Button** field, all available buttons on the joystick will be displayed (the number of buttons can be configured in the vJoy software).
- Each button indicates whether it is free or already assigned to another command.
- Select the desired button.
- The **Action** setting defines the button behavior:
  - **Press only**
  - **Release only**
  - **Press & Release**

Joystick Axis Tab



- First, select the desired joystick from the dropdown in the **Joystick** field.
- In the **Joystick Axis** field, all available axes on the joystick will be displayed (the number of axes can be configured in the vJoy software).
- Each axis indicates whether it is free or already assigned to another command.
- Select the desired axis.

Section 4 – Save/Exit

- **Save Button**: Saves the current changes and keeps the form open for further adjustments.
- **Save and Close Button**: Saves the changes and closes the form.

Cancel Button

- **Cancel Button**: Discards the current changes and closes the form.

Note: Changes made in the **Keyboard Commands** form are initially stored only in memory.
To save the changes permanently for the next program start, the user configuration must be written to the external file via the main form (click **"Save"** or **"Save As"** on the main form).

3.3. (Optional) Activating Virtual Joysticks

BMSAIT allows you to pass switch signals to the flight simulation as either keyboard commands or joystick signals. While keyboard commands can be handled via the included InputSimulator.dll library, generating joystick signals (buttons or analog axes) requires an additional setup. A joystick device must be registered in Windows to enable BMSAIT to use it for sending signals to the simulation.

If you don't want to connect a physical joystick, you can create virtual joysticks. For BMSAIT, I use the vJoy software, which is freely available at:

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

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

- **Keyboard Commands**: These are implemented via the pre-configured **InputSimulator.dll** library.
- **Joystick Signals**: To trigger joystick signals (buttons or analog axes), additional preparation is required.

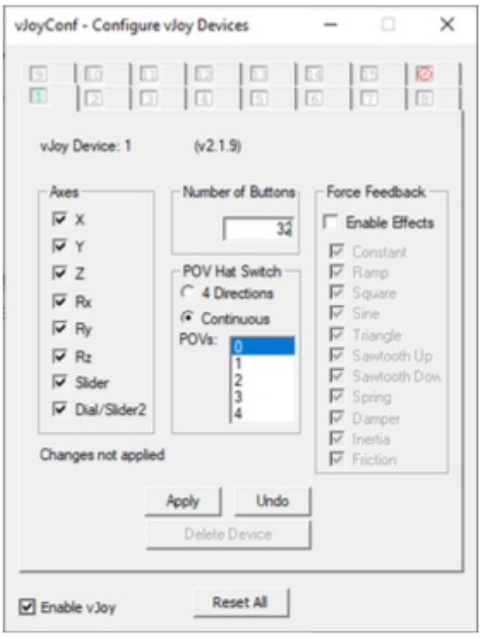
Setting Up Virtual Joysticks

Joystick signals can only be generated if a device is set up in Windows. **BMSAIT** can then use this device to send signals to the flight simulation.

To avoid connecting a physical joystick, virtual joysticks can be created using the **vJoy software**.

- Download the software for free from:

  vJoy Software Download Page



---

vJoy Configuration Steps

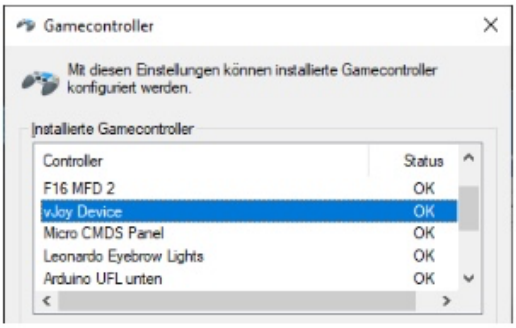
1. After installation, open the configurator **VJoyConf**.
2. Enable a joystick.
    - The default configuration is a joystick with **32 buttons** and all **analog axes** enabled.
    - **ForceFeedback** is not required and can be disabled.
3. If **more than 32 buttons** or **8 analog axes** are needed, additional joysticks can be activated:
    - Click one of the numbered tabs at the top.
    - Click the **"Add Device"** button at the bottom.

Apply Changes

- If you make any changes, you must restart Windows for them to take effect.

Under the **Game Controller Settings** in Windows, the **vJoy** device should now be visible.

If joystick processing is enabled in **BMSAIT**, it can now access the vJoy device during input processing. An input command from an Arduino can then be assigned to the buttons and axes of the vJoy device (see Chapter 3.2.5).



3.4. (Work in Progress) DCS Integration

Although BMSAIT is primarily designed as an interface for Falcon BMS, it is possible to use it with DCS (Digital Combat Simulator). This can be helpful for users who want to fly in DCS without losing the functionality of their home cockpit. Existing Arduino solutions, such as F4toSerial (BMS) and DCS BIOS (DCS), provide great functionality. However, switching between these systems typically requires reprogramming the Arduinos.

BMSAIT aims to provide seamless switching between BMS and DCS without requiring configuration changes. The BMSAIT application automatically detects whether BMS or DCS is running and processes data from either simulation. This eliminates the need to adjust Arduino programming for each simulator.

> To use this feature, the **"DCS Integration"** setting must be enabled in the basic settings, and a UDP port must be specified for receiving data from DCS.
>
> In DCS, the installation of the **"DCS-BIOS"** mod is required. This software enables the extraction of flight information. The data is transmitted via the TCP/UDP protocol and received by BMSAIT. By default, DCSBIOS sends data to port **5010**.
>
> In the `Export.lua` file of DCS, an option can be activated to allow DCSBIOS to send data to other ports as well. The specific port must be configured in the **BIOSconfig.lua** file located in the installation directory of DCSBIOS.

```
BIOS.protocol_io.connections = {
        BIOS.protocol_io.DefaultMulticastSender:create(),
        BIOS.protocol_io.TCPServer:create(),
        BIOS.protocol_io.UDPSender:create({ port = 20000, host = "127.0.0.1" }),
        BIOS.protocol_io.UDPListener:create({ port = 7778 })
}
```

Abbildung 8: Senderangaben DCS-BIOS in BIOSconfig.lua

(I personally use port 20000, as I experienced connection issues with port 5010).

Current Status:

This feature is still under development. Basic functions, such as Indexer Lights and some engine instruments, are operational. However, many additional data points need to be validated and standardized so they are properly displayed by Arduinos.

To enable this feature:

1. Activate the “DCS Integration” option in BMSAIT’s base settings.
2. Specify a UDP port for receiving data from DCS via DCS BIOS.

DCS BIOS Setup:

DCS BIOS must be installed as a mod in DCS to enable flight information extraction. It transmits data using the TCP/UDP protocol, which BMSAIT receives.

By default, DCS BIOS uses port 5010. However, in the Export.lua file of DCS, you can configure it to send data to other ports. The port must also be specified in the BIOSconfig.lua file located in the DCS BIOS installation folder.

(I personally use port 20000, as I experienced issues with port 5010.)

More information about DCS BIOS can be found at: http://dcs-bios.a10c.de/

DCS-BIOS Documentation — DCS-BIOS v0.10.0 documentation
If you are a new user, start with the next chapter: Installing DCS-BIOS.

https://dcs-bios.readthedocs.io/en/latest/

3.5. GaugeTable

In addition to the variable list, BMSAIT includes a second CSV file: `BMSAIT-GaugeTable.csv`. This file contains parameters for controlling analog round gauges (currently RPM and FTIT).

The table is necessary because the scale on some gauges is not linear; that is, the spacing between two steps on the scale is not uniform.

Example: On the RPM scale:

- Between 0° (0% RPM) and 90° (50% RPM), 50% of the RPM range is covered.
- Between 180° (80% RPM) and 270° (100% RPM), only 20% of the range is covered.

The output from SharedMem does not account for this and simply provides, for example, the raw RPM value (e.g., 65%).

The **GaugeTable** enables the pointer movements to be adjusted without requiring reprogramming of the BMSAIT app. The table specifies which angle of pointer movement corresponds to specific scale values. Users can freely define as few or as many assignments as they want. BMSAIT interpolates the pointer movements between the defined positions.

The screenshot shows the settings for the RPM scale.

Example: For an RPM value of 65%, the GaugeTable configuration would direct the pointer to position **24250**(equivalent to 133°).

	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 Arduino Program

4.1. Preparation Overview

4.1.1. Foreword

Arduinos offer an almost unlimited number of input and output device connection possibilities. It is impossible to provide a software solution that covers every option at the press of a button. Therefore, the included code serves as application examples rather than complete solutions.

When connecting a device, you must consider the interplay between:

• Arduino programming/customization

• Wiring of peripherals (e.g., correct pin assignment, device specifications, appropriate resistors for LEDs).

4.1.2. Programming an Arduino with the BMSAIT Sketch

To enable an Arduino to receive data from the BMSAIT Windows program and output it to connected devices, you must upload software (a sketch) to the Arduino. The software is provided as C++ source code.

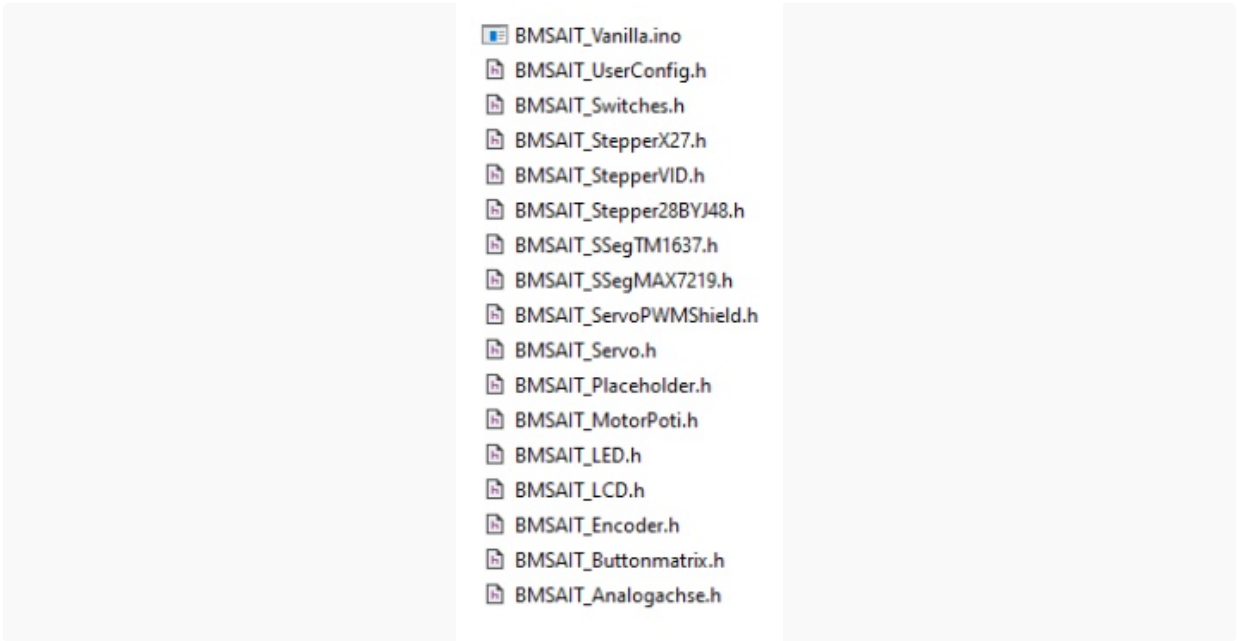
4.1.3. Downloading the Source Code

The BMSAIT code consists of several files, including:

• A `.ino` file

• Multiple `.h` and `.cpp` files

These files must be saved together in the same folder.



4.1.4. Installing a Development Environment

While these files can be viewed in any text editor, you'll need a dedicated development environment to upload them to the Arduino. I recommend the free Arduino IDE, which can be downloaded from:

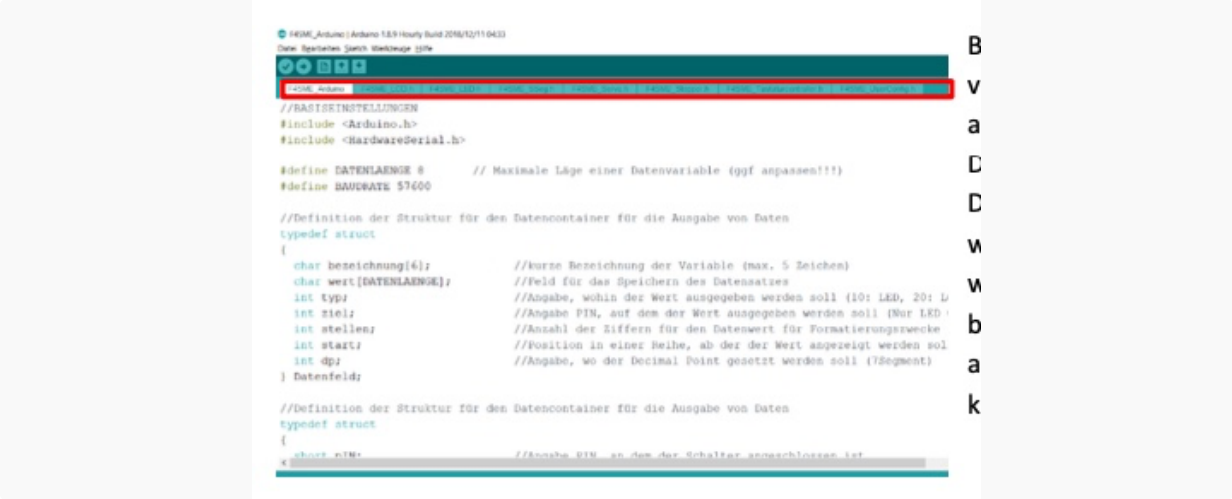
https://arduino.cc/en/main/software

Loading the Arduino Software

Once the Arduino IDE is installed, the Arduino software can be opened by double-clicking the `.ino` file.

Important: When loading the software, ensure that the various `.h` / `.cpp` files appear as tabs in the IDE. If not, the files were not placed in the same directory as the `.ino` file. In such cases, the program will likely not run.

Only the `.h` / `.cpp` files needed for your project must be included. Unnecessary module files can be safely deleted.



Installing Libraries

Before using the software for the first time, it is necessary to install additional libraries (extensions) in the Arduino IDE. The required libraries depend on the devices you wish to control via the Arduino.

Before using the software for the first time, you must install additional libraries in the Arduino IDE. These libraries depend on the devices you plan to connect to the Arduino.

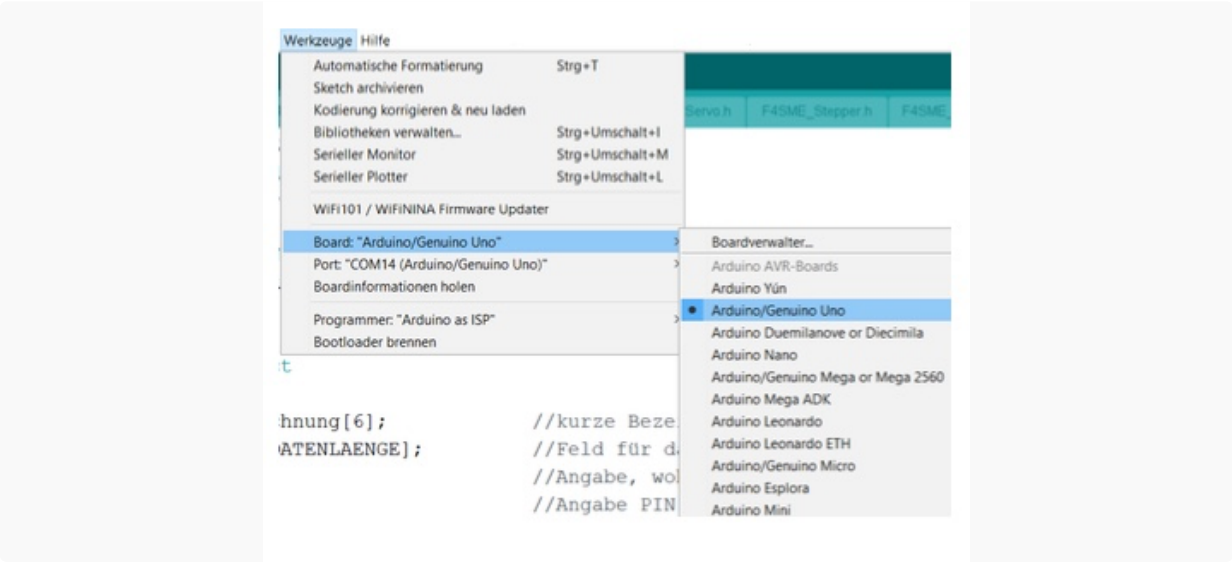
Libraries Used in the Provided Examples

Component	Library	Source
LCD (16x2 or 20x4 displays with HD44067 controller)	`LiquidCrystal_I2C.h`	LiquidCrystal_I2C
7-Segment Display (MAX7219 Controller), LED Matrix	`LedControl.h`	LedControl
7-Segment Display (TM1367 Controller)	`LEDDisplayDriver.h`	LED Display Driver
Servo Motors (direct control)	`Servo.h`	Servo Library

Servo Motor Shield (PCA9685 Controller)	`Adafruit_PWMServoDriver.h`	Adafruit PWM Servo Driver
Stepper Motors (Standard)	`Stepper.h`	Stepper Library
Stepper Motors (x27.168)	`SwitecX25.h`	SwitecX25 Library
Stepper Motor Shield (VID6606)	`SwitecX12.h`	SwitecX12 Library
OLED Displays (OLED, FFI, SBI, DED/PFL)	`U8g2.h`	U8g2 Library

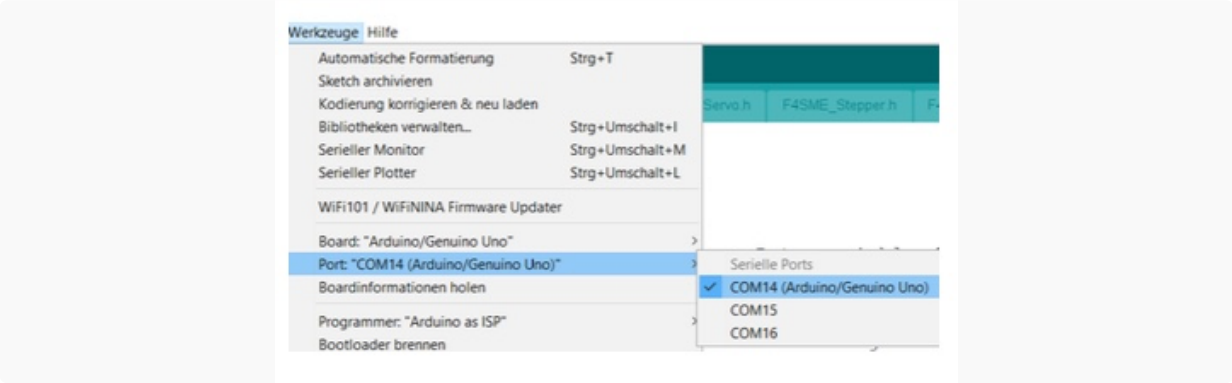
LCD (16x2 oder 20x4 Displays mit HD44067 Controller)	LiquidCrystal_I2C.h https://github.com/johnrickman/LiquidCrystal_I2C
7-Segment-Anzeige (MAX7219 Controller), LED Matrix	LedControl.h http://wayoda.github.io/LedControl/
7-Segment-Anzeige (TM1367 Controller)	LEDDisplayDriver.h http://lygte-info.dk/project/DisplayDriver%20UK.html
Servo-Motoren (direkt)	Servo.h https://github.com/arduino-libraries/Servo
Servo-Motor-Shield (PCA9685 Controller)	Adafruit_PWMServoDriver.h https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library
Stepper-Motoren (Standard)	Stepper.h https://github.com/arduino-libraries/Stepper
Stepper-Motoren (x27.168)	SwitecX25.h https://github.com/clearwater/SwitecX25
Stepper-Motor-Shield (VID6606)	SwitecX12.h https://github.com/clearwater/SwitecX25
OLED Displays (OLED, FFI, SBI, DED/PFL)	U8g2.h

4.1.5. Selecting the Arduino Board



To upload the software to the Arduino, the correct Arduino board must be selected in the Arduino IDE. This can be done via the **Tools** menu. Under the entry **"Board: xxx"**, select the type of board you are using (e.g., Arduino UNO).

4.1.6. Selecting the COM Port



Under the **"Port"** entry, select the serial port assigned to the Arduino board in Windows.

The port assigned to a board can be determined through:

- The Windows Device Manager.
- The Arduino IDE.
- The BMSAIT application.

4.1.7. Verifying the Arduino Software

The Arduino IDE automatically checks the software for errors during the upload process. If any programming errors are found, the upload process will be aborted.

I recommend verifying the software for programming errors yourself before uploading. To do this, select the **Verify/Compile** command under the **"Sketch"** menu.

When running this for the first time, errors may occur because the program code attempts to access libraries that have not yet been installed (see Chapter 4.1.4).

4.1.8. Uploading the Arduino Software

In the Arduino IDE, use the **"Upload"** command in the **"Sketch"** menu. If the correct Arduino board and COM port are selected, and the current code is error-free, the software will be uploaded to the Arduino. Afterward, the Arduino will restart and begin processing according to the uploaded program.

4.2. Description of the Arduino Sketch

The Arduino software consists of several individual files, referred to here as **modules**. This modular separation improves the clarity of the program code, as different types of devices controlled by the Arduino may require different programming logic.

By dividing the program into modules, the Arduino only loads the parts of the code required for a specific application.

In addition to the BMSAIT program and this documentation, in the repository you should find several Arduino projects. The **BMSAIT Vanilla** project serves as the foundation and should be used as a starting point for your own projects. The other included projects are examples, each providing a complete, functional implementation for common use cases.

The following description pertains to the **Vanilla Project**. Descriptions of the example programs can be found in the documentation subfolder accompanying each example program.

4.2.1. The BMSAIT Vanilla Module

In the main module (the `.ino` file):

1. The required additional modules are included.
2. Global values and variables are initialized (see Chapter 5.2.4).
3. The Arduino is set up (via the **Setup** function).
4. The main loop for continuous processing is started (via the **Loop** function).
5. Communication with the PC is managed (via the functions **ReadResponse**, **SendSysCommand**, **SendMessage**, **PullRequest**, and **DebugReadback**).

To avoid errors, no changes should be made to this module.

Exception: Modifications may be necessary if a new module needs to be added to support a device not included in the standard coding.

To integrate new types, the following steps are required:

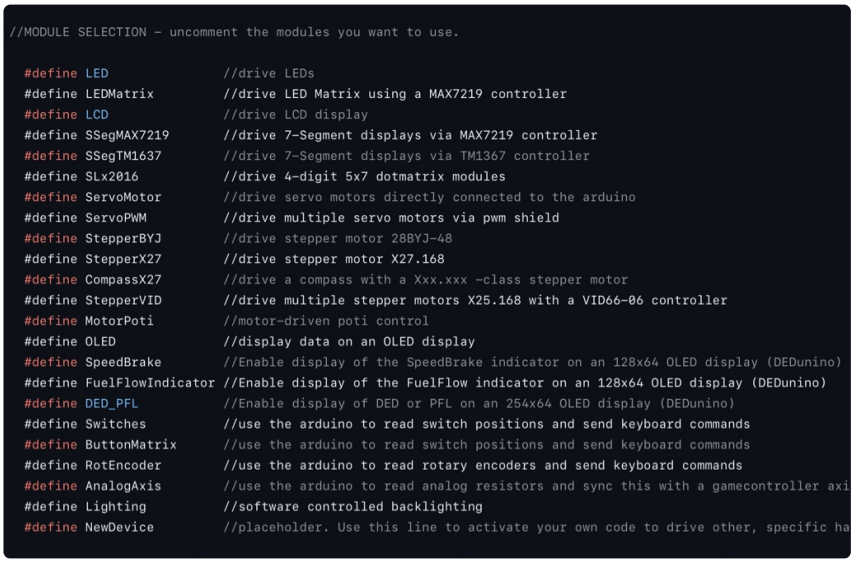
1. Include the new module.
2. Add a reference to the setup of the new module.
3. (For motors) Add a reference to the calibration of the new module.
4. Add a reference to the update logic within the main loop.

```
#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
```

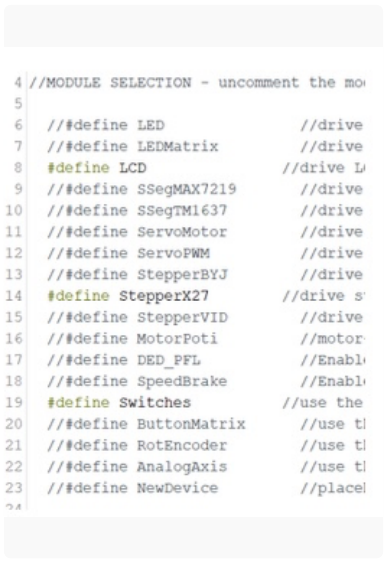
4.2.2. The UserConfig Module (`BMSAIT_UserConfig.h`)

This module consolidates essential settings that the user can or should adjust to configure the desired workflow. This base module is required for every project.

4.2.2.1. Module Selection



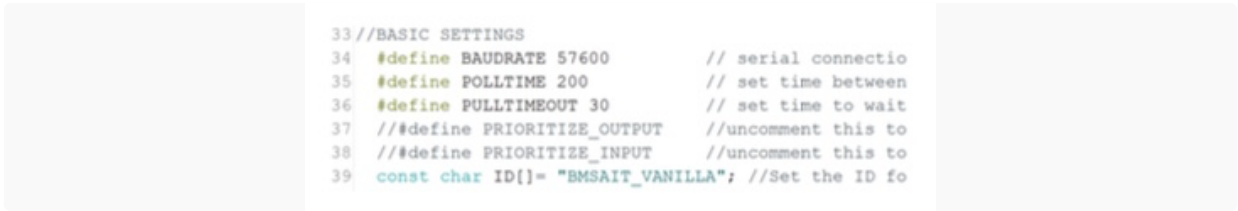
```
//MODULE SELECTION - uncomment the modules you want to use.

  #define LED               //drive LEDs
  #define LEDMatrix         //drive LED Matrix using a MAX7219 controller
  #define LCD               //drive LCD display
  #define SSegMAX7219       //drive 7-Segment displays via MAX7219 controller
  #define SSegTM1637        //drive 7-Segment displays via TM1367 controller
  #define SLx2016           //drive 4-digit 5x7 dotmatrix modules
  #define ServoMotor        //drive servo motors directly connected to the arduino
  #define ServoPWM          //drive multiple servo motors via pwm shield
  #define StepperBYJ        //drive stepper motor 28BYJ-48
  #define StepperX27        //drive stepper motor X27.168
  #define CompassX27        //drive a compass with a Xxx.xxx -class stepper motor
  #define StepperVID        //drive multiple stepper motors X25.168 with a VID66-06 controller
  #define MotorPoti         //motor-driven poti control
  #define OLED              //display data on an OLED display
  #define SpeedBrake        //Enable display of the SpeedBrake indicator on an 128x64 OLED display (DEDunino)
  #define FuelFlowIndicator //Enable display of the FuelFlow indicator on an 128x64 OLED display (DEDunino)
  #define DED_PFL           //Enable display of DED or PFL on an 254x64 OLED display (DEDunino)
  #define Switches          //use the arduino to read switch positions and send keyboard commands
  #define ButtonMatrix      //use the arduino to read switch positions and send keyboard commands
  #define RotEncoder        //use the arduino to read rotary encoders and send keyboard commands
  #define AnalogAxis        //use the arduino to read analog resistors and sync this with a gamecontroller axi
  #define Lighting          //software controlled backlighting
  #define NewDevice         //placeholder. Use this line to activate your own code to drive other, specific ha
```



```
4 //MODULE SELECTION - uncomment the mo
5
6   //#define LED               //drive
7   //#define LEDMatrix         //drive
8   #define LCD               //drive L
9   //#define SSegMAX7219       //drive
10  //#define SSegTM1637        //drive
11  //#define ServoMotor        //drive
12  //#define ServoPWM          //drive
13  //#define StepperBYJ        //drive
14  #define StepperX27        //drive s
15  //#define StepperVID        //drive
16  //#define MotorPoti         //motor
17  //#define DED_PFL           //Enabl
18  //#define SpeedBrake        //Enabl
19  #define Switches          //use the
20  //#define ButtonMatrix      //use t
21  //#define RotEncoder        //use t
22  //#define AnalogAxis        //use t
23  //#define NewDevice         //placel
```

This section defines which modules the Arduino should control.

- **Important**: Only select the modules you will actually use, as each module consumes resources that could otherwise be used for the Arduino's primary tasks.
- **To enable a module**: Remove the `//` characters before the `#define` keyword.
- **To disable a module**: Add `//` characters before the `#define` keyword.



```
33//BASIC SETTINGS
34  #define BAUDRATE 57600          // serial connectio
35  #define POLLTIME 200            // set time between
36  #define PULLTIMEOUT 30          // set time to wait
37  //#define PRIORITIZE_OUTPUT     //uncomment this to
38  //#define PRIORITIZE_INPUT      //uncomment this to
39  const char ID[]= "BMSAIT_VANILLA"; //Set the ID fo
```

```
33//8ASIC SETTINGS
34| #define BAUDRATE 57600 // serial connectio
35| #define POLLTIME 200 // set time between
```

```
36| #define PULLTIMEOUT 30| // set time to wait
37| //#define PRIORITIZE_OUTPUT uncomment this to
38| //#define PRIORITIZE_INPUT //uncomment this to
39| const char ID[]= "BMSAIT_VANILLA"; //Set the ID fo
```

```
//BASIC SETTINGS
  #define BAUDRATE 57600          // serial connection speed
  #define POLLTIME 200            // set time between PULL data requests
  #define PULLTIMEOUT 30          // set time to wait for a requested data update defaut: 30ms
  //#define PRIORITIZE_OUTPUT     //uncomment this to put a stress on fast update of outputs (should be used for motors to allow smooth
  //#define PRIORITIZE_INPUT      //uncomment this to put a stress on fast er poll of inputs (switches/Buttons)
  const char ID[]= "BMSAIT_VANILLA"; //Set the ID for this arduino program. Use any string. The program will use this ID to check in w


//BOARD SELECTION

  #define UNO         //uncomment this if this sketch will be loaded on an UNO
  //#define NANO        //uncomment this if this sketch will be loaded on an NANO
  //#define MICRO       //uncomment this if this sketch will be loaded on an MICRO
  //#define LEONARDO    //uncomment this if this sketch will be loaded on an LEONARDO
  //#define MEGA        //uncomment this if this sketch will be loaded on an MEGA
  //#define DUE         //uncomment this if this sketch will be loaded on an DUE (connected via programming port)
  //#define DUE_NATIVE  //uncomment this if this sketch will be loaded on an DUE (connected via native port)
  //#define ESP         //uncomment this if this sketch will be loaded on an ESP32 or ESP8266
```

4.2.2.2. Constants (Basic Settings)

1. **BAUDRATE**

   Sets the communication speed between the PC and Arduino.

   - The value must match the one specified in the Windows app for this Arduino.

2. **POLLTIME**

   Specifies the time in milliseconds the Arduino waits between two **PULL requests**.

   - Default: **200ms**

3. **PULLTIMEOUT**

   Specifies the time in milliseconds the Arduino waits for a response after a **PULL request** before continuing to the next request.

Prioritization

Arduinos have limited processing power, which can lead to delays or missed messages during resource-intensive tasks.

- **PRIORITIZE_OUTPUT**

  Activates additional calls to update data outputs, prioritizing fast output performance.

- **PRIORITIZE_INPUT**

  Activates additional queries for connected switches, prioritizing quick input response.

- **No Prioritization**

  If neither is activated, resources are evenly distributed between reading new data, evaluating inputs, and updating outputs.

ID

Assigns a name to the Arduino for identification by the Windows app.

- Default: **"BMSAIT_Vanilla"**

4.2.2.3. Device Type (Board Selection)

```
//BOARD SELECTION

  #define UNO          //uncommen
  //#define NANO         //uncomm
  //#define MICRO        //uncomm
  //#define LEONARDO     //uncomm
  //#define MEGA         //uncomm
  //#define DUE          //uncomm
  //#define DUE_NATIVE   //uncomm
  //#define ESP          //uncomm
```

In this section, you select the type of Arduino board being used.

- The selection is important for controlling **OLED displays** and adjusting the communication interface when using the **DUE** with its native USB connection.

4.2.2.4. Data Container (Data Variables)

```
//DATA VARIABLES

  // This is the most important part of this sketch. You need to set the data that the Arduino will have to handle
  // Make sure that you chose the definition of VARIABLENANZAHL matches the number of entries in this table
  // Fill data as follows:
  // 1. Short description (max 5 characters)
  // 2. Flightdata ID (check BMSAIT Variablen.csv)  as a 4-digit string (needed for PULL mode)
  // 3. data type (check BMSAIT Variablen.csv: b=bool, y=byte, 1=byte[], i=integer, f=float, s=string, 2=string[]))  (needed for PULL r
  // 4. Output type (10-LED, 20-LCD, 30-MAX7219, 31-TM1637, 40-Servo, 41-ServoPWM, 50-Stepper 28BYJ, 51-Stepper X27-168, 52-StepperBoai
  // 5. Target - i.e. Output PIN of LEDs, a reference to an entry in a module specific data table (i.e. motors)
  // 6. Reference2 - i.e. the line on LCD displays or a link to an entry in container in motor modules
  // 7. Reference3 - i.e. number of characters to display (control display length on 7-segment or LCD displays)
  // 8  Reference4 - i.e. start position (control position of data on 7-segment or LCD displays)
  // 9. Reference5 - i.e. decimal point (will add a decimal point on 7-segment displays after the given position)
  // 10. Initial value as string (i.e. "00")


  Datenfeld datenfeld[]=
    {
      //Description ID    DT    OT    target Ref2 Ref3 Ref4 Ref5  RQ   IV
       {"ENGFI",  "1506", 'b',  12,    1,     2,   4,   0,   0,   "",  "F"}     //Example Variable 0 - Right Eyebrow Engine Fire
      ,{"ENGFI",  "1506", 'b',  12,    1,     2,   4,   0,   0,   "",  "F"}     //Example Variable 0 - Right Eyebrow Engine Fire
    };
  const byte VARIABLENANZAHL = sizeof(datenfeld)/sizeof(datenfeld[0]);
```

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

Abbildung 9: Datencontainer

This is the **most critical configuration** you need to make.

- Here, you define which data from the **Shared Memory (SharedMem)** of BMS should be used on the Arduino and which connected hardware will display this data.
- Up to **98 data variables** can theoretically be added.

Important Note:

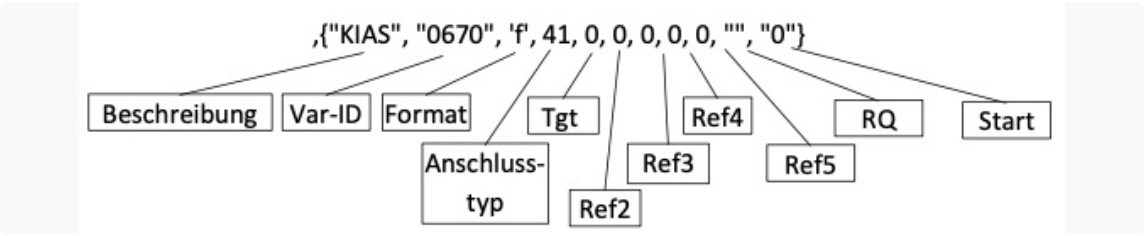
- Adding too many variables may lead to issues, as the Arduino's processing capacity is limited. This can cause display disruptions or delays.
- The exact limit of what is feasible has not yet been determined.

Syntax Tip:

- Pay attention to the **comma** at the beginning of every line from the second to the last.
- The first line of the data container does **not** include this comma.

Explanation of Each Field in a Data Container Row

The fields in each row specify how data is handled and displayed on the Arduino.



Explanation of Data Container Fields

1. **Description (Beschreibung)**
   - Informational only.
   - Can be used to display the variable name (up to **5 characters**) alongside the value on a display, e.g., an LCD.
2. **Variable ID**
   - Required only for the **PULL principle** data exchange.
   - Used by the Windows program to identify the correct data set in the **SharedMem**.
   - Enter the **4-digit number** (pad with leading zeros if necessary) corresponding to the ID from the `BMSAIT-Variablen.csv` variable list.
3. **Format**
   - Required only for the **PULL principle** data exchange.
   - Specify a key for the data format of the desired variable, as indicated in `BMSAIT-Variablen.csv`.
4. **Connection Type (Anchluss-typ)**
   - Specifies which connected device should display the content of the variable.

   ### Connection Types

   - **10**: LED (PIN is **anode**, connected through the LED to GND).
   - **11**: LED (PIN is **cathode**, connected through the LED to Vcc).
   - **12**: LED Matrix (multiple LEDs sharing Arduino control pins).
   - **20**: LCD.
   - **30**: 7-Segment Display (Max7219 controller).

- **31**: 7-Segment Display (TM1367 controller).
- **32**: DotMatrix Module (SLx2016).
- **40**: Servo (direct connection).
- **41**: Servo via PWM motor shield.
- **50**: Stepper Motor (28BYJ-48 or similar).
- **51**: Stepper Motor (X27.168, direct to Arduino).
- **52**: Stepper Motor (X27.168 via motor controller).
- **53**: Stepper Motor with compass functionality.
- **60**: Motor potentiometer.
- **70**: OLED.
- **71**: OLED with Speedbrake Indicator functionality.
- **72**: OLED with Fuel Flow Indicator functionality.
- **80**: Backlighting control.

5. **Tgt (Target)**
   - The use of this field depends on the **connection type**:
     - **Type 11 (LED)**: Specifies the pin where the LED is connected, displaying the (Boolean) value.
     - **Motors (Types 40, 41, 50, 51, 52, 60)**: Refers to an entry in the motor list. The motor lists for the respective modules contain control information for each motor (see 4.2.12).
     - **Note**: Do not provide the motor pin here; this belongs in the motor list of the corresponding module.
     - **LCD screens**: Determines the row in which the value should be displayed.
6. **Reference 2**
   - Depends on the **connection type**:
     - **LEDs**: Defines the brightness of the LED.
7. **Reference 3**
   - Depends on the **connection type**:
     - **LCD**: Specifies how many characters of the data set should be displayed.
     - Ensures correct positioning on displays like LCDs or 7-segment tubes.
     - If the data set is longer than the specified number `x`, only the first `x` characters will be shown.
8. **Reference 4**
   - Depends on the **connection type**:
     - Specifies text alignment on the output device, such as an LCD or 7-segment tube.
     - The number indicates how many positions the text should be indented to the right.
9. **Reference 5**
   - For **7-segment tubes**, this determines the position where a decimal point should appear.
10. **RQ (Request)**

- A placeholder field used by Arduinos in the **PULL principle** to prepare data request commands.

11. **Start**

4.2.3. The Switches Module `BMSAIT_Switches.h`

When this module is activated, and switches are defined in the configuration, it continuously checks whether buttons/switches are pressed. If so, commands are sent to the PC. In the Windows app, you can define which keyboard/joystick signals are triggered when a command is received (see Chapter 3.2.5).

This module assumes that all switches connect the Arduino PIN to GND.

Two types of switches can be configured:

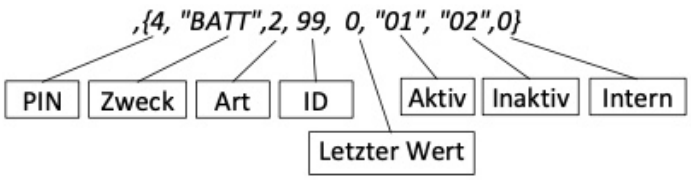
4.2.3.1. Digital Buttons/Toggle Switches

- Any number of switches can be connected to the Arduino, limited only by the number of available PINs.
- Each switch to be read must have an entry in the data block.

```
//Switch definition. If you add a switch, add a line to the following list
Switch schalter[]=
{
//  <PIN>,<description>,<type>,<rotarySwitchID>,  0, <commandID when pressed>,<commandID when released>,<internal command>
   { 2,     "MC",      1,        0,           0,          "01",                 "00",                 0}         //Example button (Master Caution pushbutton) on PIN 2
  ,{ 3,     "OVRD",    2,        0,           0,          "02",                 "03",                 0}         //Example button (Man Pitch Override) on PIN 3
  ,{ 4,     "Gear",    2,        0,           0,          "04",                 "05",                 0}         //Example switch (Gear) on PIN 4
  ,{ A0,    "UHFM",    3,        0,           0,          "00",                 "00",                 0}         //Example rotary switch (UHF Main) on PIN A0
};
const byte anzSchalter = sizeof(schalter)/sizeof(schalter[0]);
```

```
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 einer Zeile für den Schalter:



Format for a switch entry:

`{4, "BATT", 2, 99, 0, "01", "02", 0}`

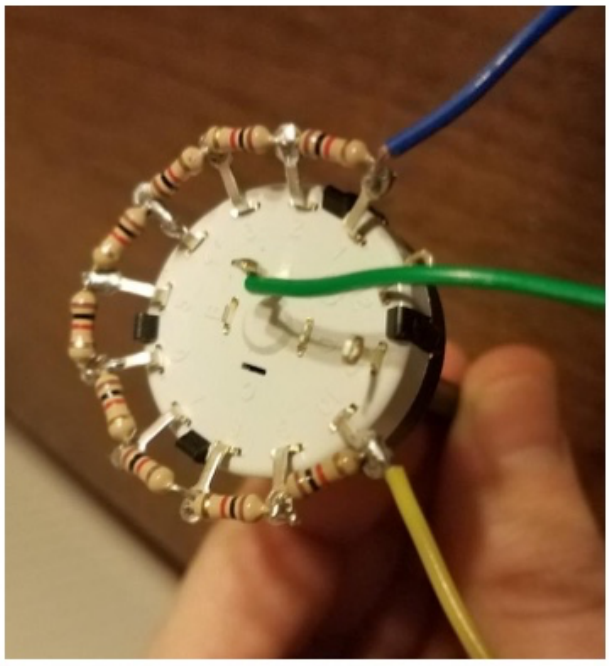
Field	Description
PIN	The PIN to which the switch is connected, used to detect changes.
Purpose (Zweck)	A brief description of the switch's purpose (optional, for clarity only).
Type (Art)	Type of switch: 1 = digital button, 2 = digital toggle switch, 3 = analog rotary switch.
ID	Identification number required for addressing blocks with multiple rotary switches.
Last Value (Ltzer Wert)	Placeholder for the last recorded value (default: 0). Used during runtime; no configuration needed.
Active (Aktiv)	Command sent to the Windows app when the switch is activated (voltage changes from 1 to 0).
Inactive (Inaktive)	Command sent to the Windows app when the switch is deactivated (voltage changes from 0 to 1).
Internal	Placeholder for internal Arduino commands (default: 0).

System Commands for Internal Use:

Buttons or switches can have internal commands for Arduino-based control. Currently, three system commands trigger synchronization with the BMSAIT Windows app (see Chapter 2.2):

- **253**: Triggers switch synchronization.
- **254**: Moves all stepper motors to their zero position.
- **255**: Calibrates all stepper motors.

4.2.3.2. Digital Rotary Switches with Analog Reading



This is a **special approach** to reading a digital rotary switch using an Arduino.

Typical Method

- Each position (detent) of the rotary switch is usually connected to a separate pin on the Arduino.
- This method can quickly use up all available pins on the Arduino.

Analog Method

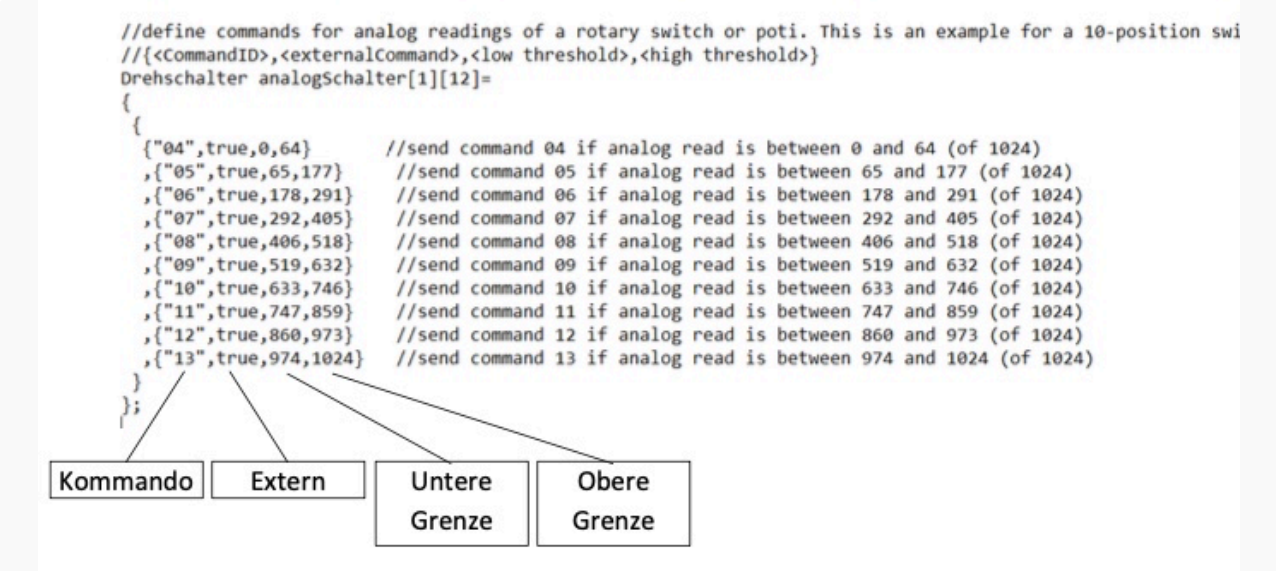
- This alternative method allows reading a rotary switch with **multiple detents** using **only one analog pin** on the Arduino.

How it Works

1. The rotary switch is wired as shown, with resistors placed between each position.
2. The movement of the rotary switch changes the number of resistors between the control signal (green) and the power supply (yellow).
3. This results in varying voltage levels, which can be read and differentiated by the Arduino's **analog input pin**.
4. The Arduino interprets these voltage levels and determines the current position of the rotary switch.
5. The measured value is output by the Arduino as a number between **0 and 1024**.

This method efficiently uses **one analog pin** to handle a rotary switch with multiple positions, conserving digital pins for other purposes. To avoid using multiple PINs for each notch of a rotary switch, this module supports reading rotary switches via a single analog PIN.

When wired as shown, the voltage read by the analog PIN changes depending on the number of resistors between the control signal and the voltage supply. By reading this voltage, the Arduino can determine the current position of the rotary switch, outputting a value between 0 and 1024.



For this to work, a control table must be created for each rotary switch.

Field	Description
Command	Command sent to the Windows app when the current notch is selected. Use "00" if no command is needed.
External	Set to "true" to send the command to the Windows app; set to "false" for Arduino-only processing.
Lower Bound	Lower threshold of the voltage range for the current notch.
Upper Bound	Upper threshold of the voltage range for the current notch.

Lower bound: When a switch is read analogically, a value between **0** (voltage at the pin equals the supply voltage) and **1024** (voltage at the pin equals ground) is output.

To account for tolerances:

- The range between **0 and 1024** is divided into as many sections as there are detents on the rotary switch.
- The **lower boundary** of the tolerance range for the current detent is specified here.
- The value should **exactly match** the upper boundary of the previous tolerance range.

Obere Grenze: When a switch is read analogically, a value between **0** (voltage at the pin equals the supply voltage) and **1024** (voltage at the pin equals ground) is output.

To account for tolerances:

- The range between **0 and 1024** is divided into as many sections as there are detents on the rotary switch.
- The **upper boundary** of the tolerance range for the current detent is specified here.
- The value should **exactly match** the lower boundary of the next tolerance range.

4.2.4. The Button Matrix Module

This module allows a matrix setup for digital switches, significantly increasing the number of switches/ toggles that can be connected to an Arduino by combining rows and columns of PINs.

- Define the PINs serving as rows and columns of the matrix.
- Create a table that specifies the signals sent to the BMSAIT app when a button/switch in the matrix is activated. Ensure the number of rows and columns matches the defined PINs.

> **Reading Digital Switches with an Arduino**
>
> When an Arduino is used to read digital switches, the limited number of available pins can quickly become a bottleneck. A solution is to use a **matrix arrangement** instead of assigning each switch to its own pin.
>
> In a matrix setup, switches close connections between pairs of digital output pins on the Arduino. For example:
>
> - With 20 available pins, the number of switches the Arduino can handle increases from 20 (direct control) to **100**using a matrix configuration.
>
> **Wiring Considerations**
>
> 1. **Button-Only Configuration**
>   - Wiring is relatively simple for buttons because only one contact in the matrix is closed at a time.

- The pressed button can be uniquely identified based on its position in the matrix.

2. **Including Toggle Switches**
   - If toggle switches are included, multiple contacts in the matrix may be closed simultaneously.
   - To properly identify which switches are active, additional measures are needed.

Solution:

- A **diode** must be placed before each button or switch.
- The diodes ensure that simultaneous activations do not interfere with the correct identification of the active switches.

```
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]);
```

Zudem müsst ihr hier eine Tabelle hinterlegen, in der die Signale angegeben werden, die der Arduino bei Registrierung eines gedrückten Tasters/Schalters an die BMSAIT Windows-App senden soll. Bitte beachtet, dass die Zahl der Spalten (col) und Reihen (row) mit der Zahl der PINs übereinstimmen muss, die ihr im vorherigen Schritt hinterlegt habt.

```
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 Encoder Module

This module supports one or two rotary encoders and their associated button (if present). Encoders can generate keyboard or joystick signals, such as for CRS/HDG knobs.

This module allows the control of one or two rotary encoders, with their movements translated into keyboard or joystick signals. These can be used, for example, to adjust the altimeter or the CRS/HDG knobs on the HSI.

The module can also read a button incorporated into some rotary encoders. This button can be used for a "shift" function, enabling a single encoder to trigger four different signals

Pin Configuration

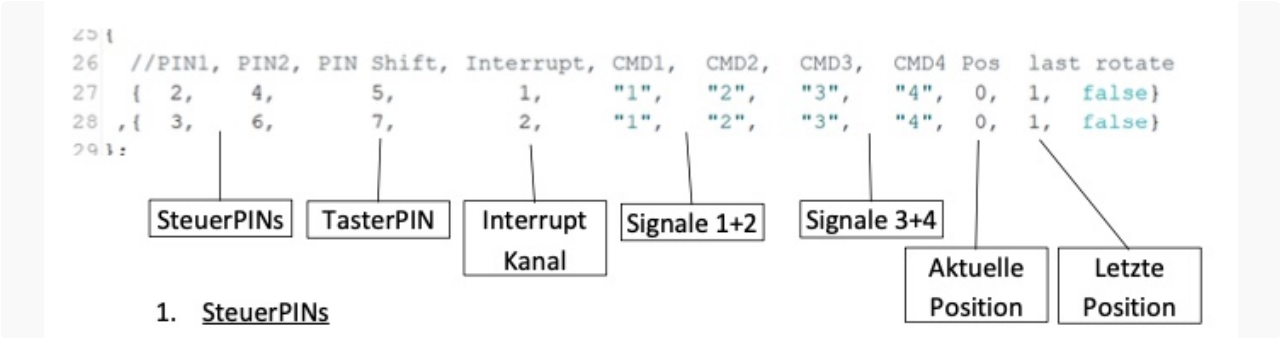
A rotary encoder typically has three pins. Refer to the datasheet of your encoder to confirm the pin assignments.

- Usually, the middle pin is connected to **GND**, and the two outer pins are connected to data pins on the Arduino.
- This module uses a special Arduino routine that enables extremely fast response to encoder movement (**Interrupts**), ensuring maximum reliability.

Interrupt Pin Requirements

Not all Arduino pins support interrupts. Each encoder must be connected to a pin capable of handling interrupts:

Board	Interrupt Pins
UNO	2, 3
NANO	2, 3
LEONARDO	0, 1, 2, 3, 7
MICRO	0, 1, 2, 3, 7
MEGA	2, 3, 18, 19, 20, 21



- **PIN Connections:** Typically, two outer pins connect to data PINs of the Arduino, and the middle pin connects to GND.
- **Interrupt Support:** Each encoder must connect to an Arduino PIN supporting interrupts (e.g., on an UNO: PINs 2, 3).

1. **Control PINs**

   Specify the two pins used to read the rotary encoder's movement (the two outer pins of the three-pin group). These pins must support **interrupts** for reliable functionality.
   - **Interrupt**: Handles fast response to changes in encoder movement.
   - **Channel**: Identifies the encoder signal being processed.
   - **Current Position**: Tracks the encoder's current position.
   - **Last Position**: Stores the encoder's previous position for comparison.
2. **Button PIN**

   Specify the pin connected to the encoder's built-in button (if available). This allows for additional functionality, such as a "shift" mode.

Signal Definitions:

- **1+2:** Commands sent when the encoder turns left/right without pressing the button.

- **3+4:** Commands sent when the encoder turns left/right while the button is pressed.

4.2.6. The Analog Axis Module

4.2.6. Das Modul Analogachse

Dieses Modul ermöglicht es, ein angeschlossenes Potentiometer auszulesen und über die BMSAIT Windows-App und der vJoy Software als Analogachse eines Joysticks abzubilden. Dies kann zur Ansteuerung einer Analogachse in BMS (z.B. TRIM, Lautstärkeregelung usw.) genutzt werden.

Hierzu müssen in dem Modul die Analog-PINs des Arduino benannt werden, die ausgelesen werden sollen:

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

This module allows reading a potentiometer and mapping it to a joystick's analog axis in BMSAIT via vJoy software.

- **PIN:** Specify the analog PIN connected to the potentiometer (e.g., A0, A1).
- **Command:** To control an analog axis in **vJoy**, a command for the analog axis must be defined in the BMSAIT Windows application. Enter the Command ID defined in the BMSAIT Windows app to link the analog data to the correct vJoy axis.
- **Value:** This field is used at runtime to store the last read value. **No modifications** are required.

Analog reading of a pin is relatively accurate, but **100% precision cannot be guaranteed**.

- The Arduino divides the voltage range of a connected potentiometer into **1024 steps**.
- Environmental factors may cause the signal on an analog pin to **slightly fluctuate**.
- To avoid triggering unnecessary signals due to minor fluctuations, a **buffer value** is defined.
- A signal is only triggered if the change in the signal exceeds the buffer value.

Recommended Buffer Value: Between **3 and 5**.

To minimize unnecessary signals due to environmental fluctuations, a buffer value (recommended: 3–5) should be set.

```
// This module allows to read analog inputs and send changes to the F4SME App
#define ATH 3   //Analog Threshold. A change of the analog value will only be co
```

If a movement of the potentiometer is detected, the read value is sent to the Windows application for further processing, where it is handled accordingly (see section 3.2.5).

4.2.7. The LED Module

This module supports simple LED setups controlled by individual Arduino PINs.

- **Type 10:** PIN provides voltage for the LED.
- **Type 11:** PIN acts as the ground connection for the LED.

Simple Control

In the simplest form of control, the program checks whether the data value assigned to the LED contains the value **"T"**(true). If yes, the LED is activated; in all other cases, the LED is turned off. No additional settings are required.

LED with Adjustable Brightness

It is possible to define the brightness of an LED directly in the Arduino programming. However, changing the brightness through the BMSAIT app is not currently supported.

Setting LED Brightness

To set the brightness of an LED, it must be connected to one of the **PWM-capable pins** on the Arduino.

The brightness is defined in the **UserConfig module** within the data field. In the row corresponding to the data variable associated with the LED, column **Ref2** specifies the brightness level, ranging from:

- **0** (off)
- **255** (full brightness).

Blinking LEDs

Some LEDs in the simulator blink under certain conditions (e.g., the JetFuelStarter light).

- The information about whether an LED is on/off or blinking is stored in different areas of the **SharedMem**.
- The BMSAIT app combines this information and transmits a combined status to the Arduino.
- This explains why some LEDs are represented as **Boolean values** and others as **Byte values** in the variable table.

Blinking Behavior:

- The BMSAIT app transmits blinking LEDs as **Byte values**:
  - **0:** Off
  - **1:** On

- **3:** Slow blinking
- **4:** Fast blinking

- The Arduino manages the blinking behavior locally (decentrally).
- If the blinking speed is too fast or too slow, it can be adjusted in the Arduino programming by modifying the **BLINKSPEED** definition.

Features:

- **Brightness Control:** LEDs can be dimmed if connected to PWM-capable PINs. Brightness is defined in the configuration with values from 0 (off) to 255 (full brightness).
- **Blinking LEDs:** LEDs can blink based on SharedMem data, with values indicating state (e.g., 0 = off, 1 = on, 3 = slow blink, 4 = fast blink). Blinking behavior can be customized in the Arduino code via `BLINKSPEED`.

To adjust the brightness of the LED, it is necessary to connect it to one of the **PWM pins** on the Arduino.

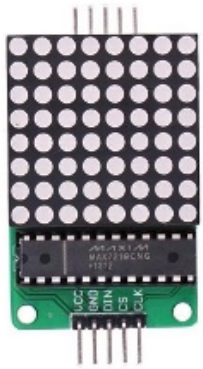
The brightness level is defined in the **UserConfig module** within the data field.

- In the row corresponding to the LED's data variable, specify the brightness in **column Ref2**.
- The brightness range is from **0** (off) to **255** (full brightness).

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

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

4.2.8. The LED Matrix Module



This module enables control of large LED arrays using a matrix setup. The recommended controller is the **MAX7219**, which connects to the Arduino via SPI.

- Define the PINs connecting to the MAX7219 in the module.
- Brightness is adjustable (`LEDM_Brightness`, 0–15).

Variable Definitions for LEDs in the Matrix:

- **Ref1:** MAX7219 chip number.
- **Ref2:** Column number.
- **Ref3:** Row number.

Preassembled MAX7219 boards with 8x8 LED matrices can be used, or individual LEDs can be connected directly to the controller for custom layouts.

In the module, you only need to specify the three pins used to 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
```

- The brightness can be adjusted using the **LEDM_Brightness** setting, with values ranging from **0 to 15**.

LED Definition in the User Configuration Module

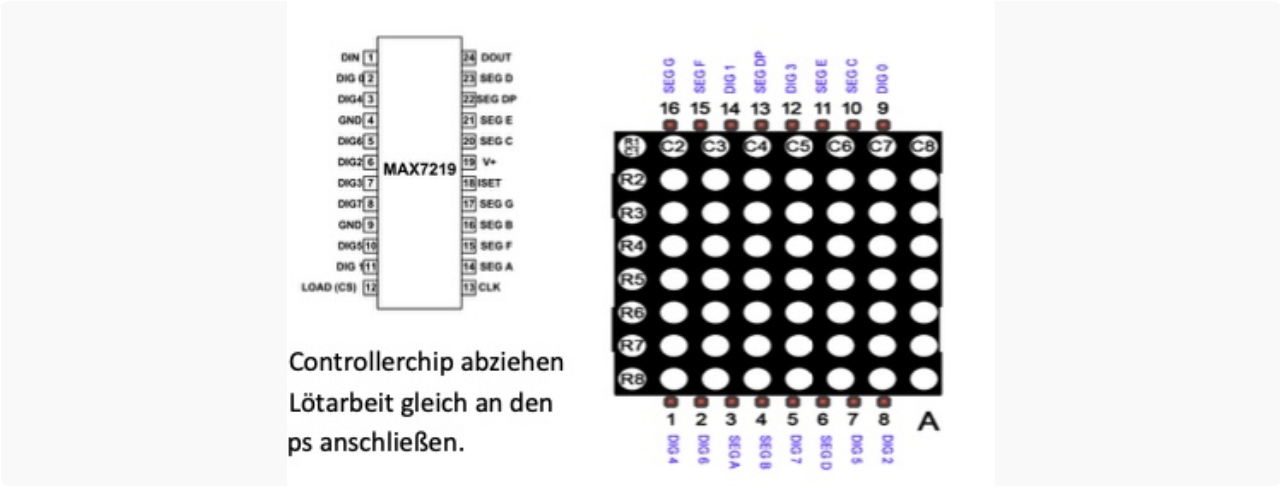
The LED is defined by specifying the appropriate variables:

- **Ref1**: Number of the MAX7219 chip (if using multiple chips).
- **Ref2**: Column where the LED should light up.
- **Ref3**: Row where the LED should light up.

Example LED Variable Definition

The following variable definition would display the **Engine Fire Warning Light (ID 1506)** on the first MAX7219 module in the **2nd column and 4th row**.

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



MAX7219 Setup and Options

- The MAX7219 chip can be purchased individually and connected.
- Pre-soldered boards with an integrated MAX7219 chip and an **8x8 LED matrix** are also available.
- If the 8x8 matrix module can be detached from the controller board, you can easily connect your LEDs to the freed-up chip outputs without additional soldering.

Matrix LED Pin Configuration

- **Segment Pins (SEG)** control the **rows**.
- **Digit Pins (DIG)** control the **columns**.
- To light up, for example, the **4th LED in the 2nd row**, a voltage must be applied between:
  - **PIN 4 (SegB = 2nd row)**
  - **PIN 12 (DIG3 = 4th column)**

LED Orientation

- The **anode (+)** connects to the **SEG (row)** output.
- The **cathode (-)** connects to the **DIG (column)** output.

LED Blinking in LEDMatrix Module

Blinking LEDs can also be implemented in the **LEDMatrix** module.

- The blinking control functions in the same way as for the **LED module**, defined through an entry in the **UserConfig module's data field**.

4.2.9. The LCD Module

This module enables the display of an alphanumeric value from a data variable on an **LCD display**. The provided coding assumes a **16x2 display (HD44780 standard)** connected via an **I2C module**.

If a different display size is used (e.g., **20x4**), the number of rows and characters per row can be adjusted in the module.

Wiring the Display

The display must be connected to the **I2C pins** of the respective Arduino board. These pins are hardware-defined, so no additional configuration in the Arduino sketch is necessary. Below are the pin mappings for various boards:

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

Displaying Multiple Variables

Multiple data variables can be displayed simultaneously on a single LCD display. This can be achieved using formatting options in the data container to place variables on different lines of the display (see section **4.2.2 / Target**).

Operating Multiple Displays

To operate multiple LCD displays on a single Arduino:

- Uncomment the code lines that reference `lcd[1]` (remove the `//` at the beginning).
- Ensure unique **hardware addressing** for each I2C device. This can be set for the LCD display using solder bridges.

4.2.10. The SSegMAX7219 Module

This module allows a **numeric value** from a data variable to be displayed on an **8-bit 7-segment display tube** using the **SPI interface** with the **MAX7219 chip**.

Configuring the Module

The pin configuration for connecting the MAX7219 display must be defined in the module. The display is connected to the Arduino using **5 wires**:

- **2 wires** for power supply (VCC and GND) are straightforward.
- **3 wires** are required for data transmission:
  - **Clock**
  - **Cable Select (CS)**
  - **Data In**

These connections must be assigned to specific Arduino pins in the module configuration.

```
#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
```

```
#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 7-segment display can be configured in the Arduino programming using the **MAX_BRIGHTNESS** option:

- **Range:** 0 (off) to 15 (maximum brightness).

  This allows you to adjust the display's brightness for various lighting conditions.

Displaying Multiple Data Variables

It is possible to display multiple data variables simultaneously on a single tube. To achieve this:

- Use the **formatting options** in the data container.

  For instance, one variable can occupy the first four digits, while another variable can use the remaining four digits.

  Refer to **4.2.2 (Character Count and Start Position)** for more details.

Decimal Point

To display a decimal point (e.g., in a radio frequency), specify its position in the data variable's configuration:

- The exact placement can be adjusted in the **Decimal Point** settings within the data container (see **4.2.2 / Decimal Point**).

This feature allows for flexible and precise formatting of numerical displays, enhancing readability and usability for applications like flight simulators.

4.2.11. The SSegTM1367 Module

This module enables the display of a numeric value from a data variable on a **3- to 6-digit 7-segment display** with a TM1637 controller chip.

Key Features

1. **Two-Pin Control**

   The TM1637 uses two data lines: **DIO** (Data Input/Output) and **CLK** (Clock). These do not rely on the I2C interface, meaning:

   - The pins do not need to be the Arduino's hardware-defined I2C pins.
   - It is not possible to use the TM1637 display alongside other devices on the same I2C pins.

2. **Pin Assignment**

   Specify the Arduino pins connected to the TM1637's **DIO** and **CLK** lines.

3. **Multiple Data Variables**

   Several data variables can be displayed simultaneously by formatting their positions:

   - Display one variable on the first few digits.
   - Display another variable on the remaining digits.

     See the **Formatting Options** section for details.

4. **Decimal Points**

   To display decimal points (e.g., for radio frequencies), specify the position of the decimal in the data variable configuration.

Example Display Configuration

The following table illustrates how to control a **6-digit display** using the data variable declarations:

Value	Target	Ref3 (Digits)	Ref4 (Offset)	Ref5 (Decimal)	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	

Brightness Adjustment

The brightness of the display can be set in the Arduino programming using the **MAX_BRIGHTNESS** option:

- **Range:** 0 (off) to 15 (full brightness).

  This allows customized visibility based on environmental conditions.

Notes

- Use **Ref3** for the number of digits to display and **Ref4** to define the starting position for the variable on the display.
- Refer to **4.2.2 / Decimal Point** for more details on setting decimal points.
- Ensure the connections for **DIO** and **CLK** are properly configured in your setup.

4.2.12 The Servo Module

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

Key Notes:

- When assigning a data variable to a motor, **do not specify the PIN directly in the UserConfig data container**.

  Instead, reference the **row in the Servo module's data container**.

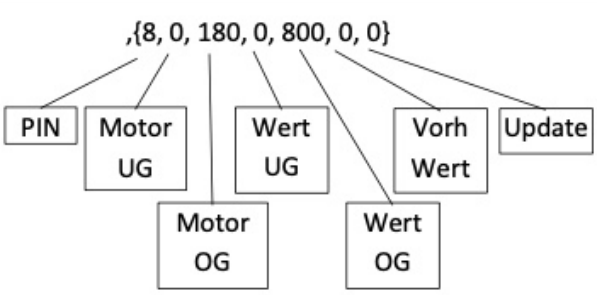
- The Servo module's data container includes both the PIN where the motor is connected and other essential operational parameters.

Configuring a Servo Motor

To control a servo motor, several configuration parameters need to be defined in the Servo module's data container. Below is the structure of a data entry:

plaintext Copiar código

```
,{8, 0, 180, 0, 800, 0, 0}
```



Field Descriptions:

1. **PIN**

   Specifies the Arduino PIN where the servo motor is connected.

2. **Motor UG (Lower Limit)**

   Defines the minimum angle the servo motor should move to, in degrees (0–180°).

3. **Motor OG (Upper Limit)**

   Defines the maximum angle the servo motor should move to, in degrees (0–180°).

4. **Wert UG (Variable Lower Limit)**

   Specifies the lowest possible value of the data variable associated with the motor.

5. **Wert OG (Variable Upper Limit)**

   Specifies the highest possible value of the data variable associated with the motor.

6. **Vorh. Wert (Previous Value)**

   A placeholder that stores the last value sent to the motor (default: 0).

   This is managed at runtime and does not require user configuration.

7. **Update (Last Signal Timestamp)**

   A placeholder that records the last time the servo received a signal (default: 0).

   This is managed at runtime and does not require user configuration.

Summary

By mapping the data variable values to servo angles, this module allows smooth and precise control of servo motors for applications like flight simulators or other automation tasks.

When calibrating a servo motor connected to an Arduino, it is important to test its **minimum (Motor UG)** and **maximum (Motor OG)** positions. During calibration, the motor will move to these limits and stay there for one second each. If the servo vibrates or produces audible strain, it indicates the motor is trying to reach a position beyond its physical limitations. While this won't typically damage the motor, it is better to avoid such situations by adjusting the **Motor UG**and **Motor OG** values.

Key Features and Challenges:

- **Limited Range**: Servo motors generally operate within a 180° range, which may not fully cover the requirements for more complex movements in flight instruments.
- **Absolute Positioning**: A significant advantage of servo motors is their ability to move directly to an exact position (e.g., "move to 90°") without needing to know their previous position.

Functionality of the Servo Module:

The **Servo module** maps the current value of a data variable (e.g., flight simulator instrument data) to the servo's physical position. This is done by scaling the data variable's range (`Wert UG` to `Wert OG`) to the servo's movement range (`Motor UG` to `Motor OG`).

Example:

- If the data variable equals the **minimum value (`Wert UG`)**, the servo moves to its **minimum angle (`Motor UG`)**.
- If the data variable equals the **maximum value (`Wert OG`)**, the servo moves to its **maximum angle (`Motor OG`)**.
- For a value in the middle of the range, the servo moves to the midpoint of its movement range.

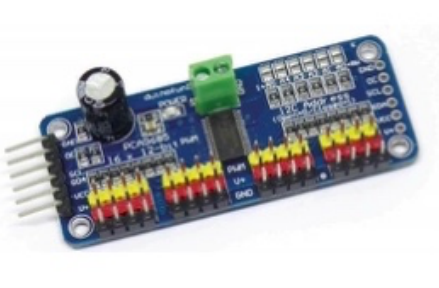
Practical Steps for Setup:

1. **Calibrate the Servo**:
   - Ensure the servo doesn't vibrate or strain at the defined limits (`Motor UG` and `Motor OG`).
   - Adjust these values as needed for smooth operation.
2. **Set Data Variables**:
   - Define the range of the incoming data (`Wert UG` to `Wert OG`).
   - Map this data to the servo's movement range for synchronized operation.
3. **Test for Synchronization**:
   - After calibration, test the motor's responsiveness to changes in the data variable.
   - Fine-tune the values for smooth transitions and accurate positioning.

Benefits of the Servo Module:

- **Precise Control**: The module ensures accurate mapping of data values to servo positions.
- **Efficient Integration**: The ability to define ranges and calibrate motors reduces wear and maximizes compatibility with simulation instruments.

4.2.13 The ServoPWMShield Module



This module allows the control of one or more servo motors connected to an Arduino via a motor shield, such as the **PCA9685**.

Key Points:

- **Calibration**: Servos often have slight manufacturing differences, even between identical models. Each servo should be calibrated by checking its minimal and maximal positions to avoid unnecessary strain or vibration. Adjust the `Motor UG` (lower limit) and `Motor OG` (upper limit) values accordingly during setup.
- **Functionality**: Servos controlled through this module can be precisely positioned without needing to track their previous positions, making them ideal for simulating flight instruments.

Servo Calibration

- During calibration, the servo moves to its defined **minimal** and **maximal** positions, remaining at each position for one second. If the motor vibrates or strains during this process, the position cannot be physically achieved. Adjust the `Motor UG` and `Motor OG` values to eliminate strain.

Core Operation

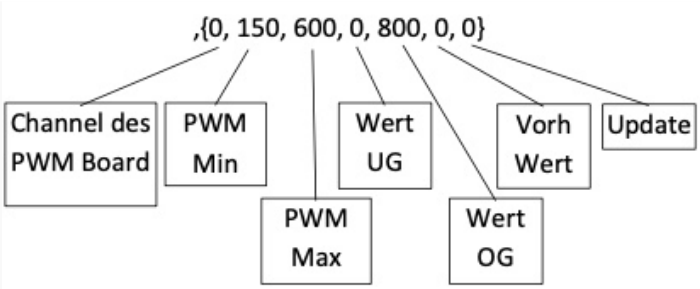
- **Position Mapping**: The module calculates the servo's position based on the input data variable. For example:
  - If the data variable equals the defined minimum value (`Wert UG`), the motor will move to its minimal angle (`Motor UG`).
  - If the data variable is midway between its minimum and maximum values, the servo moves to the midpoint of its range.

Servo Data Configuration in ServoPWMShield

To control a servo motor, several parameters must be defined in the **ServoPWMShield module's data container**. Here is the structure of a data entry:

plaintext

Copiar código



`,{<ShieldAddress>, <Channel>, <Motor UG>, <Motor OG>, <Wert UG>, <Wert OG>, <LastValue>}`

Field Descriptions:

1. **Shield Address**

   The I2C address of the PCA9685 shield connected to the Arduino.

2. **Channel:** The specific channel on the motor shield to which the servo is connected.
3. **Motor UG (Lower Limit):** The minimal angle the servo can achieve (0–180°).
4. **Motor OG (Upper Limit):** The maximal angle the servo can achieve (0–180°).
5. **Wert UG (Data Variable Lower Limit):** The minimum value of the associated data variable.
6. **Wert OG (Data Variable Upper Limit):** The maximum value of the associated data variable.
7. **Last Value:** Placeholder for the servo's last position. This is managed at runtime and does not require configuration.

Advantages of ServoPWMShield:

- **Extended Channels**: The PCA9685 shield allows for up to 16 connected servos, expanding beyond the Arduino's native PWM limitations.
- **Simplified Wiring**: I2C communication reduces the number of required connections between the Arduino and servos.
- **Independent Power Source**: The shield can handle external power for servos, preventing strain on the Arduino.

This module ensures smooth and precise control of multiple servos for dynamic simulations like flight instruments or complex automation tasks.

Every motor is different, and even identical servos may have manufacturing tolerances, which is why calibration is necessary. During the initialization of the Arduino, the motor will move to its minimum and maximum positions, staying in each position for one second. During this time, check if the Arduino vibrates perceptibly or audibly. If it does, the servo is trying to reach a position that cannot be achieved due to hardware limitations. While this likely won't damage the servo, it should generally be avoided. Therefore, the **PWM Min** and **PWM Max** values should be checked and adjusted as needed for each connected motor.

Servomotors have the disadvantage that they cannot move freely; they are limited to a small range of movement for an arm or pointer (usually 180°). However, the major advantage of servos is that the arm or pointer can be controlled absolutely, meaning you don't need to know the previous position of the pointer to move it to a desired position (→ move to position 90°).

The functionality of the **Servo** module involves converting the current value of the data variable into the position the motor should move to. To achieve this, the data variable is compared with the values **Wert UG** (lower value) and **Wert OG** (upper value). If the data variable's value equals the defined minimum value (**Wert UG**), the motor will also move to its minimum position (pulse duration **PWM Min**). If the data variable's value is exactly midway between the minimum and maximum values, the motor will also move to the midpoint between the minimum and maximum positions.

The motor can be damaged if you attempt to move it to positions outside the permissible angular range (usually 180°). Therefore, it's crucial to ensure that the control information provided is configured correctly to prevent this.

4.2.14. The Stepper Module

This module is used to control one or more stepper motors (e.g., **28BYJ-48**), each connected to the Arduino via a motor control shield (for the **28BYJ-48**, the **ULN2003** shield).

Stepper motors have the advantage of being able to move freely without encountering physical limits. However, the downside is that the motor/software does not know the pointer's position when the program starts. Since the control is relative (→ move 20 steps to the right), the previous position of the pointer must be known to move it to a desired position. This means that **a stepper must be manually or automatically calibrated at program start t**o ensure the arm/pointer can subsequently be moved to the correct position.

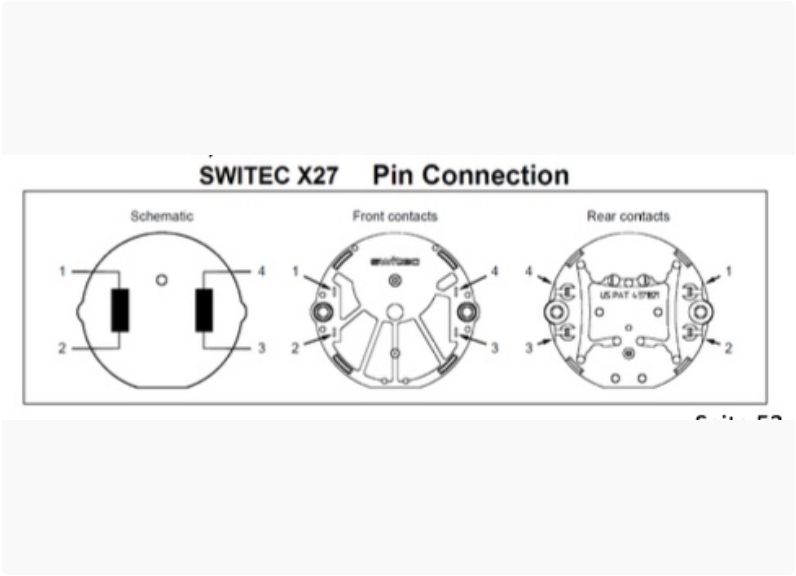
If the stepper is used for a display with physical limits (e.g., a speed indicator), the position can be determined by triggering microswitches at the limits. Once the switch is triggered, the pointer's position is known and can be reliably calculated from there.

For stepper motors with low torque, another possibility is to simply restrict movement (e.g., block the indicator needle at the lower limit with a physical stop) and have the motor perform a full rotation. Some stepper motors (e.g., **x27-168**) already have such stops built in.

For displays where the needle must move freely (e.g., altimeters, compasses), the display must either be manually set, or a light barrier can be used, which is triggered only at a specific position of the display.

Currently, the BMSAIT distribution does not include a working example demonstrating the functionality of a stepper motor in conjunction with flight information from BMS via BMSAIT. This will be provided in a later version.

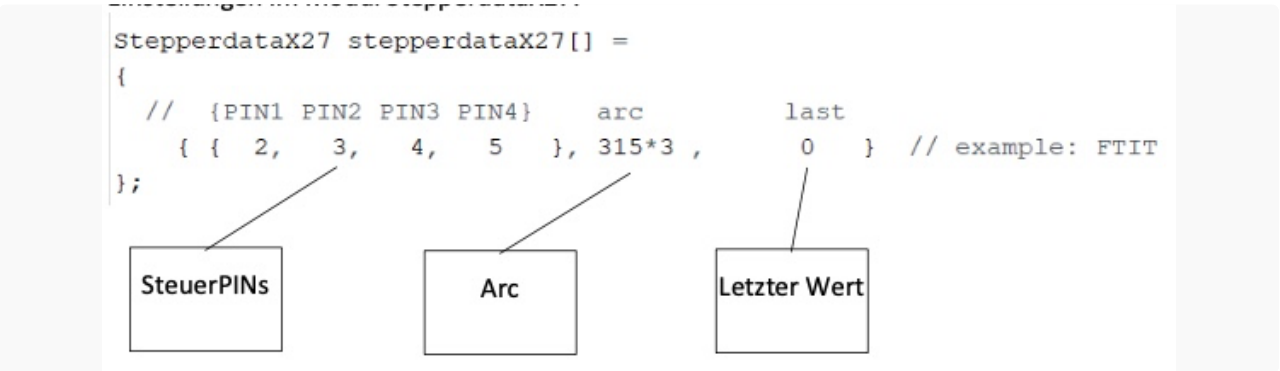
4.2.15. The StepperX27 Module



This module is designed to control the **X27.168 stepper motor**. A key advantage of these motors is that they cannot rotate freely (360°) but are limited to a maximum angle of 315°. However, this hardware limitation is not a disadvantage. The motor's restricted movement, combined with its low torque, allows it to be deliberately "overdriven" at its limits, placing it in a defined state. This enables the software to determine the pointer's position and subsequently position it accurately.

As a result, the X27 motor can almost be considered a servo motor, but it offers a greater range of motion compared to typical servos, which reliably cover only 180°.

The **X27 module** expects a dataset from the **BMSAIT app** that represents the motor's current position as a number from **0**(full deflection left) to **65535** (full deflection right). The raw data from FalconBMS must be transposed by the Windows app. Additionally, non-linear transformations (e.g., FTIT or RPM scale display) are performed by the app (see **3.5**).



Settings in the StepperDataX27 Module:

1. **Control PINs**

Specify the four pins to which the motor is connected. Ensure the correct order of the wires with the motor's terminals. Incorrect wiring will cause the motor to behave unpredictably.

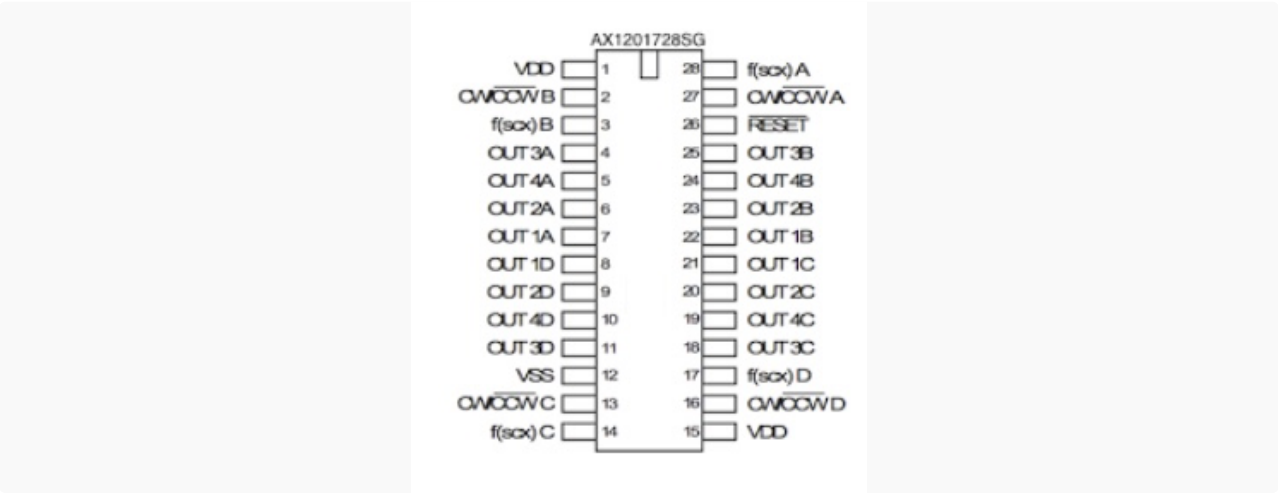
2. **Arc**

   Define the number of steps the motor can take between its limits. The accuracy of the X27.168 motor is approximately **3 steps per degree** over an arc of **315°**. If the motor needs to represent a display with a smaller angle than 315°, this can be achieved by reducing the step count.

3. **Last Value**

   This field stores the last known raw data value. It is only required during runtime and does not need to be manually changed.

4.2.16. The StepperVID Module



This module is designed to control multiple stepper motors of the **X27.168** type or similar variants (e.g., the **X40** with two concentric axes). A key advantage of these motors is their restricted movement to a **315° arc** instead of a full **360° rotation**. This limitation, combined with their low torque, ensures that the motor can reliably reach a **defined state** when it hits its mechanical stops. Knowing the motor's position enables the software to correctly move a display to its target position. The X27 motor can therefore be considered almost like a servo motor but with the added advantage of a wider range of motion compared to typical servos, which are generally limited to **180°**.

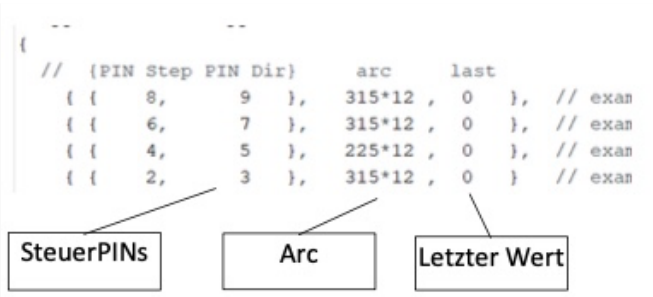
Each X27.168 motor requires **four pins** for control, which may exceed the capacity of a single Arduino. Additionally, the Arduino's power supply may not be sufficient to reliably control multiple motors.

VID6606 Controller Chip

The **VID6606** controller addresses these limitations. One chip can control up to **four motors**, reducing the pin requirements (to just **two pins per motor**) and taking over the power supply for the motors.

A diagram of the VID6606 pin configuration is shown alongside for reference.

The **StepperVID module** expects a dataset from the **BMSAIT WinApp** that represents the motor's current position as a number from **0** (full deflection left) to **65535** (full deflection right). The raw data from FalconBMS must be processed by the Windows app. The app also performs a non-linear conversion (e.g., for FTIT or RPM scales).



Arduino-Coding Settings in the StepperVID Module

1. **Control Pins**

   Specify the two pins through which control signals for a motor are sent to the VID6606. The first pin controls the desired rotation direction, while the second pin issues the movement command.

2. **Arc**

   Define the number of steps the motor can take between its mechanical limits. The VID6606 offers a precision of **12 steps per degree** over the motor's 315° range. For displays with a smaller arc than 315°, this can be adjusted by reducing the step count.

3. **Last Value**

   This field stores the last known raw data value. It is only required during runtime and does not need to be manually adjusted.

4.2.17. **MotorPoti Module**

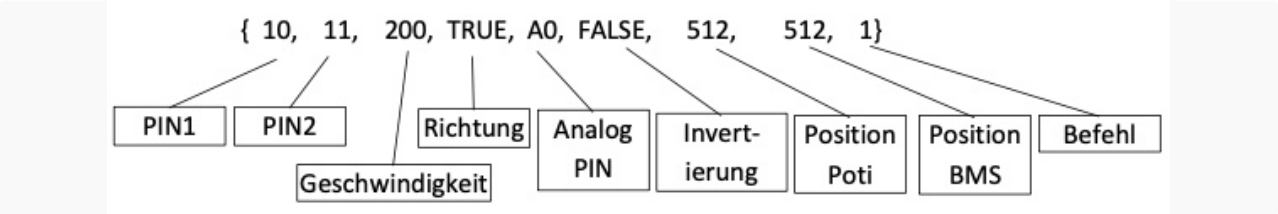
This module is designed to control one or more motorized potentiometers, where both reading and transmitting an analog value and adjusting the potentiometer's position using a motor are possible (e.g., a trim knob on the trim panel).

The software continuously compares:

- The potentiometer's analog value,
- The position of an analog axis in Falcon BMS, and
- The resulting simulation effects (e.g., trim value).

If a discrepancy is detected, it determines whether the change originated from the simulation or manual adjustment of the potentiometer. The other value is then aligned accordingly.

The motor of the **MotorPoti** must be connected to the Arduino via a motor shield (H-bridge). The **HG7881** controller board is recommended.



Configuration of MotorPoti in the Module

For each motorized potentiometer, add an entry to the block `Struc_MotorPoti motorPoti[] = {}`.

Example entry:

c

Copiar código

```
{10, 11, 200, TRUE, A0, FALSE, 512, 512, 1}
```

Field Descriptions

1. **PIN1**

   The pin controlling the motor's rotation direction (**A1**, **B1**, **C1**, or **D1** on the HG7881 board).

2. **PIN2**

   The pin triggering the motor's rotation command (**A2**, **B2**, **C2**, or **D2** on the HG7881 board).

3. **Speed**

   Controls the motor's speed by setting a pulse width (value range: **0 to 255**).

   - If connected to PWM-compatible Arduino pins (e.g., Uno/Nano: **3, 5, 6, 9, 10, 11**), speed is fully adjustable.
   - For non-PWM pins, use a minimum value of **124**.

4. **Direction**

   Saves the motor's last rotation direction.

5. **Analog PIN**

   The pin where the potentiometer's signal pin is connected.

6. **Inversion**

   Set to **FALSE** by default. If the analog axis in BMS is inverted, set this to **TRUE**.

7. **Position Poti**

   Stores the potentiometer's last position value (**0 to 1024**).

8. **Position BMS**

   Stores the value the simulation expects (**0 to 1024**).

9. **Command**

   The command ID linking the potentiometer with a vJoy analog axis in the Windows app (see **3.2.5**, section on "Joystick Axis").

Notes

- Ensure motorized potentiometers are calibrated for smooth operation.
- The comparison logic ensures synchronization between physical hardware and the simulation.

MotorPoti Module: Automatic Synchronization

The **MotorPoti** module attempts to automatically synchronize:

- The motor's movement direction,
- The resulting potentiometer position changes,
- The vJoy analog axis, and
- The axis mapping in Falcon BMS.

Troubleshooting Synchronization Issues

If the module encounters issues, it might be necessary to adjust the wiring:

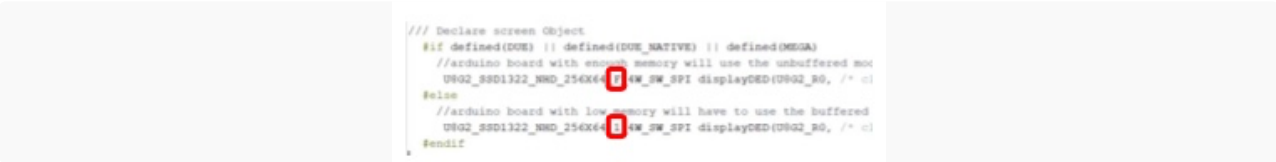
1. Swap the **+/– connections on the motor**.
2. Swap the **+/– connections on the potentiometer**.

These adjustments can help resolve inconsistencies in synchronization between the hardware and the simulation controls.

4.2.18. OLED Module

The **OLED module** is designed to display simple information on an OLED screen. The primary challenge is configuring the Arduino to properly control the display. With various OLED sizes, controller chips, vendors, and connection types available, pre-configuration is essential. This is achieved by defining a **constructor** at the beginning of the sketch, containing all relevant details for proper communication.

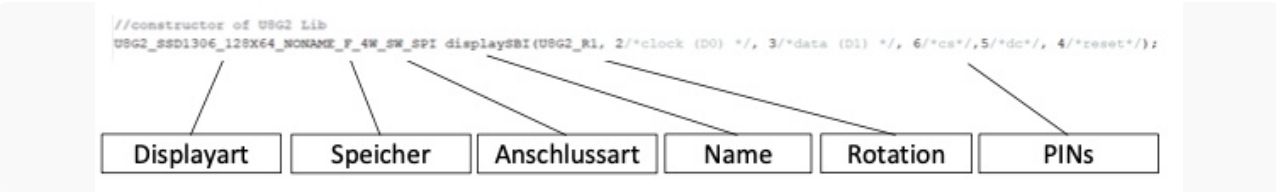
Memory Management Block



Controlling an OLED can consume significant Arduino resources. The code includes a memory management block that adjusts based on the Arduino type defined in the **User Configuration**. When using a custom display and replacing constructor lines, ensure the memory management is set to **"F"** in the first block and **"1"** in the second block.

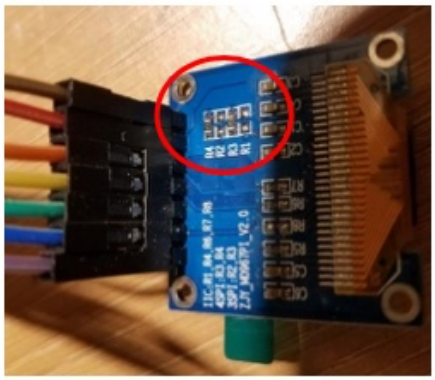
U8G2 Library Constructor

This module, along with the **Speedbrake Indicator**, **Fuel Flow Indicator**, and **DED/PFL modules**, relies on configuring the OLED. The U8G2 library handles the display, requiring information about the specific OLED type through a constructor. A full list of constructors is available here.



The constructor defines:

1. **Display Type**: Specify the display's controller chip and size. Copy the appropriate entry from the U8G2 constructor reference list.
2. **Memory Buffering**:
   - Full Buffering (`F`) for systems with sufficient resources (e.g., Mega, Due).
   - Partial Buffering (`1` or `2`) if memory constraints arise.



3. **Connection Type**: Options include:
   - 2-wire I2C,
   - 3- or 4-wire SPI,
   - 8-wire parallel.

   A 4-wire SPI is recommended where possible (e.g., `4W_SW_SPI`).

   Ensure the OLED's connection type is configured correctly via solder bridges on the OLED board.

4. **Name**: Define the variable name for referencing the OLED in the Arduino code.
    - **OLED**: `displayOLED`
    - **SBI**: `displaySBI`
    - **FFI**: `displayFFI`
    - **DED/PFL**: `displayDED`
5. **Rotation**:
    - `U8G2_R0`: No rotation.
    - `U8G2_R1`: 90° CW.
    - `U8G2_R2`: 180°.
    - `U8G2_R3`: 90° CCW.
6. **Pins**: Specify the Arduino pins connected to the OLED.

Layout Adjustment

The module allows layout customization. Text positioning can be adjusted using parameters like **OFFSETX**, which typically shifts text rightward (if the display isn't rotated). A reference table is included to guide parameter adjustments based on the display's rotation.

```
//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
```

The **OLED module** provides flexibility for displaying information, with adjustable layout and memory usage, making it suitable for a variety of projects involving different display configurations.

4.2.19. Speedbrake Indicator Module

The **Speedbrake Indicator Module** is designed to display the Speedbrake Indicator (SBI) on an OLED screen. This module is largely based on the **DEDuino project** (https://pit.uriba.org/tag/deduino/) but has been adapted for communication with BMSAIT and some display-specific controls.

Depending on the Speedbrake position, the OLED will show:

- The text **"CLOSE"** for a closed Speedbrake,
- A graphic representing the open Speedbrake (3x3 circles),
- Or the **"OFF"** display for inactive states.

OLED Requirements

Small displays (~1 inch) with a resolution of **128x64 pixels** are recommended. The code assumes the OLED uses the widely available **SSD1306 controller chip**. Other displays can be used but may require adjustments (refer to constructor settings in **4.2.18**).

A) U8G2 Constructor

Set up the constructor as described in **4.2.18**.
Additionally, configure the display dimensions using the **SB_SCREEN_W** and **SB_SCREEN_H** options (default: **128 width x 64 height**).

B) Layout 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
```

While the default constructor settings should suffice, fine adjustments can be made for precise control of the OLED display.

1. **OFFSETX and OFFSETY**: Shift the display horizontally and vertically to align it within the panel frame.

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

2. **SBIDELAY**: Time (in milliseconds) the "OFF" display remains active after Speedbrake movement (default: **250ms**).
3. **ANIMATION**: Controls whether transitions between **OPEN** and **CLOSED** are animated:
   - **Disabled**: The "OFF" flag is shown as a static image during transitions.
   - **Enabled**: The display scrolls between modes.

     Use **OFFSET_FRAMES** to adjust the number of animation frames. Higher values produce smoother animations but depend on the Arduino's performance and the OLED's refresh rate.

Module Behavior

- The OLED is inactive by default:
  - If **BMSAIT** on the PC is not active, the display enters **standby mode** after 10 seconds.
  - If **BMSAIT** is active but BMS/DCS is not in the 3D environment, the **"OFF"** display is shown.
- When the aircraft's power is off, the **"OFF"** display is shown.
- **Animation Disabled**: The **"OFF"** display briefly appears during any Speedbrake movement.
- **Speedbrake Closed**: Displays **"CLOSED"** for angles under 1°.
- **Speedbrake Open**: Displays **"OPEN"** for angles above 1°.

This module offers precise control and flexibility to visualize the Speedbrake status, ensuring it integrates seamlessly into the simulation cockpit.

4.2.20. Fuel Flow Indicator (FFI) Module

The **Fuel Flow Indicator (FFI) Module** is designed to display the Fuel Flow value on an OLED screen. This module uses a data variable formatted by the **BMSAIT Windows App** (IDs **0511/0521**) to present the information in the correct format. The module is adapted from **DEDuino** (https://pit.uriba.org/tag/deduino/) for compatibility with **BMSAIT** and includes additional display controls.

A) U8G2 Constructor

The OLED constructor settings are described in **4.2.18**.
Ensure to set the display dimensions with **SB_SCREEN_W** and **SB_SCREEN_H** (default: **128x64 pixels**).

B) Font Settings

- The module uses a font specifically configured for the FFI display.
- This font supports only the necessary characters: **numbers**, and the letters required for "FuelFlow" and "PPH."
- Other characters cannot be displayed.
- Font size is defined by **FF_CHAR_W** and **FF_CHAR_H**; these settings should not be changed.

C) Layout Settings

These settings allow fine-tuning of the Fuel Flow value's position on the OLED display. Adjustments may be necessary to align the text perfectly within a panel or bezel.

D) Options

The module includes two additional features that enhance the FFI display. By default, these options are enabled for Arduino boards with sufficient performance. To disable them, comment out the respective settings by adding `//`.

1. **REALFFI**: Adds animations to the digits when the Fuel Flow value changes.
2. **BEZEL**: Draws a frame around the display with static text "Fuel Flow" and "PPH."

Module Behavior

- **Inactive Mode**: The OLED remains off if there is no connection to **BMSAIT**.
- **Connection Established**:
  - When FalconBMS is not in the 3D world, the display shows a **zero value**.
  - While in the 3D world, the last valid Fuel Flow value is displayed.
  - If the connection leaves the 3D world, the display returns to showing a **zero value**.
- **Timeout**: The OLED turns off if no data is received from **BMSAIT** for 10 seconds.

This module ensures accurate and visually appealing Fuel Flow data representation, fully synchronized with simulation states.

4.2.21. The DED/PFL Module

With BMSAIT, OLED displays can also be controlled for various purposes. OLED displays come in many different sizes and with various controller chips, making it impossible for this module to provide a comprehensive solution for all types of OLEDs.
This module is based on **DEDuino** (https://pit.uriba.org/tag/deduino/) but has been adapted in several areas for use with BMSAIT.

A) The U8G2 Constructor

The OLED display is controlled using the Arduino **U8G2 library**. To properly call the library and use the OLED, a constructor must be defined in the header of the module. The required settings for this are described in Chapter 4.2.18.

B) Options

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

The way the DED is controlled does not allow for custom layout settings. The only options available are:

- **PRE_BOOT_PAUSE** and **POST_BOOT_PAUSE**, which specify the time in milliseconds that the DED displays the test image during startup.

Functionality

This module enables the DED or PFL to be displayed on a 254x64 OLED screen.

- The DED and PFL are displayed as 5 lines of 24 characters each.
- At a resolution of 254x64 pixels, this results in a character size of **12x10 pixels**.

Important considerations:

- A font must be installed for text to be displayed on the OLED.
- The font has a fixed size for each character, so it is necessary to know in advance how large the displayed information will be to load the appropriate font.
- Due to the Arduino's limited resources, it is only possible to include a very limited number of fonts.

The DED module includes a font saved as C code in the file **FalconDEDFont.h** (see Appendix 5.2.5).

- This font contains only the characters required for displaying the DED (lowercase letters and some special characters are missing) and is not suitable for other purposes.
- Each character is assigned a code that generally matches the ASCII code but deviates for certain DED-specific characters.

Data Handling:

- The DED data is stored in two variables in the **SharedMem**:
  - **230 "DED"**
  - **245 "INV"**
- However, these data are not easily readable because the DED includes special characters not found in the standard character set (e.g., inverted text, cursor arrows).
- The **BMS Windows App** reads the data from the SharedMem and modifies the bytecode before sending it to the Arduino.
- The Arduino receives this data as correctly mapped characters based on the predefined character set and displays it directly.

Module Behavior

1. The display is initially off. Without a connection to BMSAIT, the display remains inactive.
2. If a data connection with BMSAIT is active, but Falcon BMS is not in the 3D world, a test value is displayed.
3. While the PC is in the 3D world, the current DED/PFL content is displayed.
4. If the PC exits the 3D world, the test value is displayed again.
5. The display turns off if no data is received from BMSAIT for 10 seconds.

5. Attachments

5.1. Sources / References

- **F4SharedMem.dll** by LightningViper
  https://github.com/lightningviper/lightningstools
- **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/
- **BMSAIT Icon**
  Created by Ahmad Taufik, downloaded from thenounproject.com.
- **Code components for the DED and FuelFlow modules**
  Adopted from the DEDuino project by Uriba:
  https://pit.uriba.org/tag/deduino/
- **Code for X27 stepper motors and VID6606 chip control**
  From Guy Carpenter, retrieved from:
  https://guy.carpenter.id.au/gaugette/2017/04/29/switecx25-quad-driver-tests/

5.2. Field Descriptions

5.2.1. Data Variables (`BMSAIT-Variablen.csv`, Windows App)

- **ID**:
  A unique identifier for a data variable. This ID is mapped to a specific SharedMem section of BMS in the program code. Changes or additions to IDs require developer intervention.
- **Group**:
  A user-defined categorization of data variables for sorting purposes only.
- **Format**:
  Defines the type of value a variable can hold and determines allowable operations. SharedMem data exists in different formats, and BMSAIT must handle each accordingly.
  Incorrect format assignments in the Windows App or Arduino can lead to issues.

Data Type	Full Name		Range
Byte	Byte	v	0x00 – 0xFF (decimal: 0–255)
Float	Float	i	-3.4_10³⁸ to +3.4_10³⁸
Integer	Integer	f	-32,768 a 32,767
Text	String	s	Arbitrary characters
Bool	Bool	b	True (T) / False (F)
Byte group	Byte[]	1	Multiple Bytes
Integer group	Int[]	2	Multiple Integers
Text Group	String[]	3	Multiple Strings
Decimal Group	Float[]	4	Arbitrary decimal numbers

Note: Types such as `ushort`, `uint`, and `uint[]` exist in `BMSAIT-Variablen.csv` but are not currently supported. Contact the developer for potential integration.

- **Type**:

  Indicates the data format of the variable, used by BMSAIT as processing guidance and included in every data packet sent to the Arduino.

- **Designation**:

  A short description, usually derived from the F4SharedMem library.

- **Description**:

  A more detailed explanation of the variable's contents, also typically from the F4SharedMem library.

5.2.2. Variable Assignment (Windows App)

- **Formatting String**:

  Controls the structure of the data packet sent to the Arduino. Must include the character `@`, which represents the placeholder for the transmitted data value. Characters before or after `@` are prepended or appended to the data value during transmission.

  - Default: `<@>` (compatible with standard BMSAIT Arduino software).

- **Test Data**:

  Used to send data to connected Arduinos in **test mode**, even without a running Falcon BMS connection.

- **Position**:

  Specifies the row in the Arduino's data container where the transmitted value will be written.

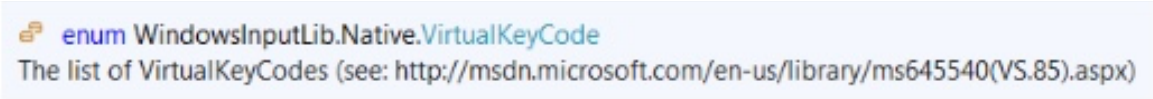
  - Must be unique for each variable.
  - **Cannot** be 99. Values above 100 are only needed for specific cases like the backup radio configuration.

- **Update Frequency**:

  Controls how often unchanging data values are re-sent to the Arduino to minimize errors.

5.2.3. Commands (Windows App)

- **ID**: A unique identifier for the command. When transmitted by an Arduino, this triggers the corresponding action (e.g., keystroke or joystick signal).
- **Type**: Specifies the type of command:
  - (1) Keyboard press.
  - (2) Joystick button press.
  - (3) Joystick axis adjustment.
- **Joystick**: Identifies the vJoy joystick used to send the signal.
- **Key/Button**:
  - **Keyboard Command**: Defines the keypress as a virtual command using the VirtualInput library.



  - **Joystick Button**: Specifies the joystick button number.
  - **Joystick Axis**: Specifies the index of the joystick axis (0–9) based on the vJoy HID_Usages list.

- **Ctrl/Alt/Shift**: Indicates whether the Control, Alt, or Shift keys are included with the command.
- **Mode**: Specifies whether the key is pressed, pressed and released, or only released.

5.2.4. Global Variables (Arduino)

- **BAUDRATE**:

  Must match the Windows App baud rate for communication. Default: 57600.
- **DATENLAENGE**:

  Specifies how many characters the Arduino can store per data value. Default: 8 (7 usable + 1 termination character).
- **MESSAGEBEGIN**:

  Byte indicating the start of a new message. Must be unique. Default: 255.
- **HANDSHAKE**:

  Byte that triggers a response with the Arduino's ID when received. Default: 128.
- **SWITCHPOSITION**:

Byte that triggers sending input states (e.g., toggle switches). Default: 150.

- **CALIBRATE**:

  Byte that initiates motor calibration for stepper motors. Default: 160.
- **ZEROIZE**:

  Byte that resets motors to zero position. Default: 161.
- **STARTPULL**:

  Byte that starts PULL processing. Default: 170.
- **ENDPULL**:

  Byte that stops PULL processing. Default: 180.
- **TESTON**:

  Byte that enables debug mode to return test information to the Windows App. Default: 190.
- **TESTOFF**:

  Byte that disables debug mode. Default: 200.
- **VAR_BEGIN/VAR_ENDE**:

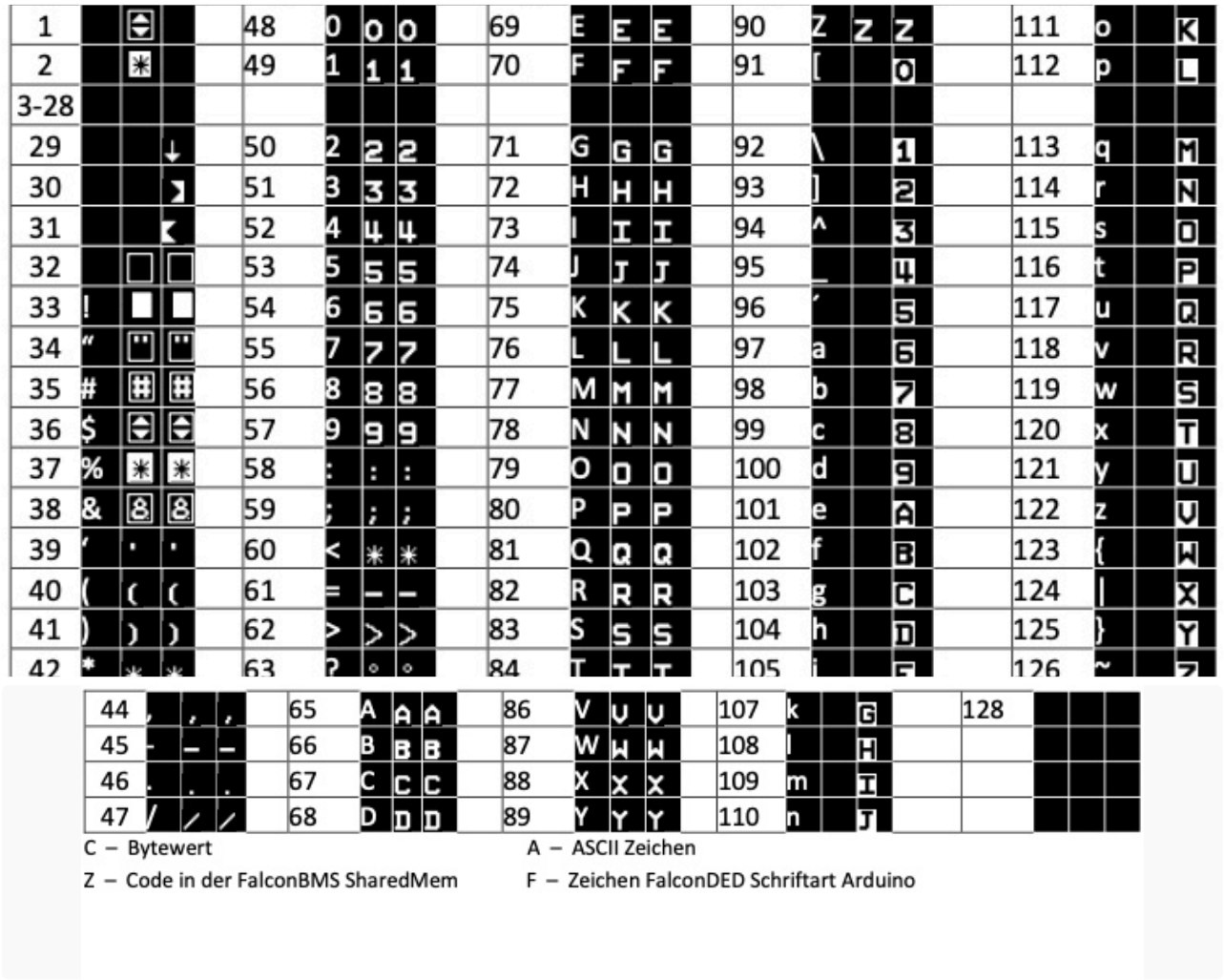
  Characters marking the start `<` and end `>` of a transmitted data value.
- **TYP_ANFANG/TYP_ENDE**:

  Characters marking the start `{` and end `}` of a data type.

Parameter	Description	Default
BAUDRATE	Must match the Windows App baud rate for communication.	57600
DATENLAENGE	Specifies how many characters the Arduino can store per data value (7 usable + 1 termination).	8
MESSAGEBEGIN	Byte indicating the start of a new message. Must be unique.	255
HANDSHAKE	Byte that triggers a response with the Arduino's ID when received.	128
SWITCHPOSITION	Byte that triggers sending input states (e.g., toggle switches).	150
CALIBRATE	Byte that initiates motor calibration for stepper motors.	160
ZEROIZE	Byte that resets motors to zero position.	161
STARTPULL	Byte that starts PULL processing.	170
ENDPULL	Byte that stops PULL processing.	180
TESTON	Byte that enables debug mode to return test information to the Windows App.	190
TESTOFF	Byte that disables debug mode.	200

VAR_BEGIN/ VAR_ENDE	Characters marking the start `<` and end `>` of a transmitted data value.	`<` / `>`
TYP_ANFANG/ TYP_ENDE	Characters marking the start `{` and end `}` of a data type.	`{` / `}`

5.2.5. Font DED/PFL



The font used for the DED and PFL is stored in the file `FalconDEDFont.h` as C code. The font was created specifically for use with the DED and PFL displays and therefore contains only the characters needed for these purposes.

- **Omitted characters**: Lowercase letters and some special characters are missing because they are not used in the DED display.
- **Font size**: Each character has a fixed size, defined for proper rendering on the OLED display.
- **ASCII compatibility**: Each character is assigned a code, which generally corresponds to its ASCII value. However, some DED-specific characters deviate from the standard ASCII codes.

This custom font ensures that all DED/PFL data is displayed correctly on the associated OLED display. For details about the font layout or adding additional characters, see the source file in the provided documentation.