Documentation ModLab V3.0

for the modular laboratory station ModLab

Made with LaTeX
Compiled: April 2, 2023

MODLAB

Synthron

admin@synthron.de

Contents

I. List of Tables

II. List of Figures

1. Introduction

The ModLab is a modular electronics workstation designed from scratch and built with mostly readily available parts.
The ModLab is the successor of my Arduino Workstation which was meant to be a better version of the FMAS (***F****lexibles* ***M****odulares* ***A****usbildungs-****S****ystem*, Flexible Modular Training System) I made during my apprenticeship at the Karlsruhe Institute of Technology. The backplane on the FMAS consisted entirely of AC voltages and one 9V DC voltage which is not really ideal. Also it was all pure logic with no programmable parts. This made it quite functional but stupid (not dumb, but no programmable parts), so I decided to improve on it.
The Arduino Workstation was my first attempt and (from my current perspective) served only as a proof of concept. My skills were quite low at that time and all I knew was Arduino, so I went with that. It kinda worked, but was not really useful and since I made some bad design choices, I had to scrap it. But it was a start and laid out a foundation.
With the ModLab I wanted to finally build my dream system. With the motto “absolutely over-engineered” it is supposed to be powerful, better planned and more intelligent. Also the 19“-Rack houses the FMAS as well, to show the roots of it (and also I didn’t want to scrap it because of nostalgia).
V2 (V1 is the Arduino Workstation) isn’t really documented, because a) it was utter crap and b) it didn’t work. I used an Arduino DUE board as main controller which did the job, but not really well. . .
The current version is V3 and is under development. I uses primarily STM32 controllers, but still has the capability of using other controllers as well.

Before I start talking about all the modules, I want to talk a bit about the FMAS and the Rack itself, so bear with my ramblings and enjoy reading.

2. The Rack

As I said in the introduction, the ModLab actually consists of two parts. To keep it simple, I will do the following: If I speak about the ModLab, I mean the ModLab itself in the upper 6U. The rack itself (which will be explained here) and the FMAS are talked about seperately.
So now let's talk about the rack.

2.1. The Idea

When I sat down at the drawing board (ok, I admit, it was Excel...) and the first thoughts in mind had, I quickly decided to make everything just bigger. The reason was, that just one 3U rack was a bit too small (e.g. the FMAS rack only had 6 slots, but 12 available modules). Coincidentally I got my hands on a 20U 19"-rack which should be plenty for my use. But now there was a new issue: next to being big and heavy, I needed to include an electric installation for the subsystems. But this will be tackled a bit later.

2.2. Planning it out

Planning out the sections of the rack was done in Excel, so no big calculations for you here, but oh well...
So I had 20U to spend and I had a lot to put in there. Thankfully, the rack has a front and a back side. For convenience I decided the "user side" would be only on the front, the installation and support systems would be on the back. The first thought was that the whole front would consist of 3U subracks, but that would be a bit much and I still needed the control circuits as well as cable channels, power supplies etc.
I decided to dedicate two 3U subracks to the ModLab and the FMAS respectively, this would reserve 5U for the control panel and cable management. This way, I can save up space on my workbench by storing the FMAS as well as having plenty of space for the ModLab modules.
After a bit of time, I came up with the following. I am quite happy how it turned out, so this will hopefully last until the end of it.

U	Front	back
20	ModLab	Cover Plate
19		Power Supplies
18		24V & 5V
17	ModLab	Cover Plate
16		Power Supplies
15		15V & -15V
14	Control Panel	Cover Plate
13		Filter
12		Fuses
11	DIN-Rail	DIN-Rail
10		
9		
8		
7	Cable Channel	Cable Channel
6	FMAS	Cover Plate
5		Voltage Regulator
4		Fuses
3	FMAS	Cover Plate
2		Mains Connector
1		Transformer

Table 2.1.: Rack Overview

2.3. Construction

So I ordered everything I needed and took some time to draw schematics, drill out the cover plates, assemble everything and soon I was done.
Well, if only it was that easy. . .
I took my time for every step just to avoid big errors. I made the electric schematic with the software “QElectroTech” and even of this schematic, there are many revisions.
With the plan done, I finally knew what devices and how many terminal blocks I needed. With this in mind I was able to prepare the cover plates and assemble them. In the meantime I designed the backplane for the ModLab, so I could already order them and assemble the subracks.
The control panel is used to enable power to the different parts of the rack as well as being home to the most important big red “OH SHIT”-button. This way, only the parts that will be used are powered. The main control voltage is enabled by a seperate switch for a bit more extra safety.
The switched power supplies for the ModLab made me some headaches. According to their datasheets they can use up to 40A at 230V in the moment of power-up. Since I got four of them in parallel, I was worried about the 160A current spike in the mains blowing the fuses in my fusebox. So I got a current limiter which is actively limiting the current during the first 70ms of power-up and then switching over to unlimited. This will definitely save my fusebox as well as the cables in the wall (and not to forget the contacts in the relais making the connections).
The transformer for the FMAS is the original one used for it, so the old rack for it got completely disassembled.

While the power lines for the FMAS are soldered directly to the backplanes of the subracks, I decided to use high-current DIN 41612 H15-connectors for the ModLab. This way I can partially disassemble the rack for transportation or make it easier to make changes or upgrades to the rack or subracks. Also the connectors are made for use in 3U subracks, so easy to implement and with the use of cable slugs really modular.

3. FMAS

Now let's talk about the FMAS, what it is and why I keep in in my inventory.

3.1. Background

The FMAS, or ***F**lexibles **M**odulares **A**usbildungs-**S**ystem* (Flexible Modular Training System), was part of my apprenticeship at the Karlsruhe Institute of Technology (KIT). It was used to teach how to design schematics, how to draw layouts and how to make PCBs (yes, we etched our own PCBs). Furthermore we were taught how to use measurement equipment, since once a PCB didn't work, the trainer basically just told us "here are multimeter and oscilloscope, find your error". As rude as that sounds, it made us think about the circuit, trace the signals around and see where we made the error.
The FMAS itself was built in a 3U rack with full enclosure, including the transformer, fuses and a regulator. We trainees built them ourselves from the provided kits.
The further along we were with training, the more modules we had for it. During vacation when vocational school was closed or when we had nothing better to do (which happened to me a lot. I was a big nerd and quite fast compared to my fellow trainees) we made the modules for the FMAS, the documentation for it (after 4 or 5 modules, we had quite a good template for it, so that wasn't too much work) and then asked for the next project.

3.2. The System

The FMAS itself was not up-to-date or highly integrated. The backplane bus consisted of 9V DC as well as 12V, 24V and 20V-0-20V AC voltages. All the other available lanes were unconnected and unused. So it got me thinking.
The modules themselves were completely discrete, so no programmable ICs or microcontrollers. A big advantage in case of there were no software bugs and to be honest, the circuits just *worked*. Also it was not necessary to contemplate life choices over forgotten semikolons in the code while debugging for days with not much knowledge about programming. It just made life to our trainers and for us way easier.
The disadvantage I see on the concept itself. Nearly only AC voltages on the backplane means that we had to rectify, filter regulate the voltage on nearly every module. Also there was lots of space on the backplane to make something interesting. With relatively low effort one could connect the modules over serial interfaces with one another and make something interesting and a little bit more intelligent and up-to-date.
But as the motto goes: "never change a running system", so my trainers were not willing to change it.

3.3. Modules

The modules themselves are nothing special, but I will give you a list of them here.

- Power Supplies: simple power supplies with LM317 and LM337 as well as one with LM723
- LED-Tester: Current Source and Voltmeter
- Waveform Generators: Discrete (fixed frequency) in different designs as well as a X2206CP-function generator
- Signal Tracer: a square wave generator with integrated speaker
- Logic Probe Tester: shows the logic level on a trace, simple discrete circuit
- LED-dice: your standard 1-6 dice...
- Continuity Tester: basically the same as the Signal Tracer
- other useless stuff like thermometers etc...

As you can see, some are useful, others are just for fun.
In the end I decided to discard the more useless modules because they really are just your basic “My First Electronics Project” kind of type. . .

4. ModLab

Okay. As I already said, I didn't like the idea of a completely discrete system. During my second year of training (it was 3 years total) I was already done with all the FMAS modules and my instructors didn't really know what to do with me. So I proposed to develop my own system and thus the "Arduino Workstation" was born. I made a LOT of bad design choices back then which resulted in it being labeled a proof of concept. After that I wanted to make a better one with the though of "overengineered and thought out as much as possible" in mind. Obviously I don't plan *everything* in advance for years and then build everything at once in the end, but I definitely put more thought into designing it and keeping it as modular as possible. Also I try to avoid bottlenecks and major design errors by thinking not only about the module itself, but the I always have the whole system in mind.
And because of this, I decided to make it bigger. Not only in dimensions, but also in effort. The ModLab is my long-term project which will grow and improve over time with no fixed deadlines and goals.
And that is how this project came to be.

4.1. Lessons Learned from the Arduino Workstation

4.1.1. No virtual Grounds!

A huge error was my thought, that +12V and -12V can be used as 24V. With some effort this surely is possible, but it includes a whole lot of difficulties and things to consider.
So why bothering with that? Just use a fixed 24V rail on the backplane and boom, done.

4.1.2. No hand-wired backplanes!

I don't think I have to talk a lot about that...
The backplane of the Arduino Workstation was handwired with all 32 pins across several connectors. Suffice to say that I had some connection issues and it was a work I don't want to do again. Making a PCB is definitely easier and well designed way more modular.

4.1.3. Planning, Planning, Planning

The old system was just made up as I went. Nothing was considered, I just wanted to play around.
Now everything is planned through, difficult circuits are simulated first and every design will get a review. This will help avoiding major errors and hopefully ensures the success.

4.2. What will be different?

First of all, the planning. This part will be a more elaborate and thorough, which is why this project will take some time. Also I want to digitalize a lot more. In the Arduino Workstation, I still used potentiometers and buttons as interfaces and regulators, so a lot of hands on manipulation. This will be changed with digi-pots as well as DACs or PWM.
Also I went away form Arduino. Actually I scrapped AVRs as a whole. Most things will use a more powerful STM32 controller to help with timing and speed.

Nonetheless maybe some AVRs or PIC microcontrollers will find there way into the system. Not for things that need a lot of processing power, but nostalgia can go a long way and if they fit the function, why not.

5. The Bus

The Bus, or more specifically the backplane bus, is connecting all the modules together. It has all necessary supply voltages as well as all traces for the IO. For this reason, this part took a while.

5.1. Voltages

In order to keep things simple but functional, I decided to plate 4 voltages on the bus. What they are and there uses are outlined below.

5.1.1. 5V

I don't think this needs a lot of explanation. Even tho most modules will run on 3V3, I might want to use other controllers or peripherals which run on 5V. For the most part, I will use Low-Dropout regulators to go to 3V3, even tho there will be quite a few of them. The Power Supply for the 5V can provide up to 15A, but I will put a 10A fuse in line. I think this will be more than enough for my uses.

5.1.2. 24V

What would be a big project without decent power? Correct: BORING. So, especially for bigger power supplies, I included 24V. This will have a 10A fuse as well, I don't think I will need much more for most projects. And if I will have some really power-hungry ideas in the future, they should have an integrated power supply anyways.

5.1.3. Symmetric 30V (-15V..+15V)

Ok, some of you might ask "what the hell is that even used for???"
Hear me out. Even tho I am way more comfortable with digital circuits, there exists something called analog circuits. And especially for these I might need these voltages a lot. Examples would be amplifier circuits, waveform generators as well as different IO interfaces.
Also I want to explore more power supply designs for negative voltages, this can come in handy.

5.2. Interfaces

The modules need to talk to each other, therefore there have to be some interfaces on the bus. To keep it modular, I decided to include several different ones. These are explained in a bit more detail here.

5.2.1. I2C

I2C (or $\mathrm{I^2C}$, IIC, *Inter Integrated Circuit* etc..) is one of my favourite interfaces, because it is simple, easy to implement and I worked with it a lot. It supports up to 128 different nodes which is more than plenty and doesn't need a whole lot of chip-select signals.
Actually I implemented two I2C interfaces, one at 100kHz standard frequency and one at 400kHz.
Even tho it is a widely used interface for tons of peripherals, it will be mainly used as backup or for small periodical data packets. This will keep the main interface more free for other uses.

5.2.2. RS485

RS485 is included, because I want to explore a protocoll called ModBUS. This might come in handy for other projects. Maybe this will be used more later, for now it is just there to have it.

5.2.3. SPI

SPI (Serial Peripheral Interface) is a really common bus for a lot of things. It can go to quite high data rates, but also needs a chip-select for every node. For this reason, some dedicated IO lines are used as Chip Selects. If they are used as-is or if I decode them is still subject to debate.
Also I actually have two SPI interfaces on the bus. one at 5V and one at 3V3. This will help integrating different controllers or peripherals to the system without a lot of level-shifting and thus introducing delays and synchronization issues.

5.2.4. CAN

The CAN (Controller Area Network) interface will be the main workhorse interface of the system. It can support a lot of nodes and due to it being a differential interface should work pretty reliably.
The downside is, that it needs a transceiver for every node but oh well.
My current plan is to use the ISO-TP protocol for it. This way I can send more than a maximum of 8 bytes per data transfer, but this might slow down the interface a bit. Because of this, I will operate the interface at 1Mbit/s datarate.

5.3. I/Os

Of course I will need some general IOs on the bus to control different things. Here a small summary.

5.3.1. Digital

There are 6 general IOs on the backplane. These can be used as status signals (error, interrupt) or to do some fun things I still have to come up with.
Also there are 6 dedicated Chip-Selects for the SPI interfaces. This way I can selectively address them.
All IOs are 5V compatible.

At first I wanted to include some form of slot address encoding, but yeah... that idea was scrapped pretty early because I would need quite a few pins for that.

5.3.2. DACs

Two DAC-channels from the Host Bus Controller (more on that later) are also on the bus. My plan is to use them with a speaker module to make really annoying system notification, like startup-melodies, error-notifications, acoustic signals for different events from submodules etc.
Let's see what will come out of that.

5.4. Layout

Ok, now that we have talked about what signals will be on the backplane, let's see how it is layed out.
I settled with the DIN 41612 64pin A-C connectors. These will fit well with the Rack-design as well as provide enough pins for everything.
I decided to give the supply voltages 4 pins each to maximize possible current draw. Also I placed some GND-pins in the middle as well. This left only a few pins for all the signals.
here is what I came up with.

	a	c	
5V	1	1	5V
5V	2	2	5V
GND	3	3	GND
GND	4	4	GND
24V	5	5	24V
24V	6	6	24V
GND	7	7	GND
GND	8	8	GND
CAN+	9	9	CAN-
IO0	10	10	IO1
SDA_FM	11	11	SCL_FM
IO2	12	12	IO2
SDA	13	13	SCL
IO4	14	14	IO5
RS485+	15	15	RS485-
GND	16	16	GND
GND	17	17	GND
DAC0	18	18	DAC1
CS0	19	19	CS1
MOSI1	20	20	MISO1
CS2	21	21	CS3
SCK1	22	22	SCK2
CS4	23	23	CS5
MOSI2	24	24	MISO2
+15V	25	25	+15V
+15V	26	26	+15V
GND	27	27	GND
GND	28	28	GND
-15V	29	29	-15V
-15V	30	30	-15V
GND	31	31	GND
GND	32	32	GND

Table 5.1.: Bus-Layout

6. ModLab Modules

This chapter is **Work in Progress** and thus subject to change at any time!
Here I wanna give you a brief overview over the modules (no particular order). Each one will be discussed in more detail, so no worries.
I will also mark everything that is currently just a concept or idea with a (c) behind its name.

- Bus Adapter
  - 1:1 Extension of the backplane bus with connector at the front
  - Termination resistors for differential pairs
  - LED indicators for supply voltages
- Host-Bus-Controller / HBC
  - Brain of the ModLab
  - Master for all backplane interfaces
  - 5” TFT LCD with touch
  - ESP32 Co-Processor
  - SD card slot for logging and more
  - USB interface
- Symmetric Linear Power Supply
  - Symmetric regulated Voltage
  - Both CC and CV modes
  - Controlled via PWM
- Symmetric Switching Mode Power Supply (c)
- Linear Power Supply (c)
- Switching Mode Power Supply
  - High-current power supply
  - Both CC and CV modes
  - Controlled via PWM
- Waveform Generator Analog (c)
- Waveform Generator DDS
  - AD9833 Waveform Generator
  - 0.1Hz to 12.5MHz
  - Sine wave, triangle and square wave

  - VCA and Output Driver
- Logic Probe Tester (c)
- Diode Tester
  - 4 different constant current sources
  - capable of creativ characteristic curves
  - Co-Prozessor übernimmt Regelung
- Electronic Load (c)
- Speaker Module (c)
- Oscilloscope

6.1. Bus Adapter

The Bus Adapter is one of the first modules. It's main function is to terminate the differential bus lanes. For this reason the Bus Adapter is made twice to terminate each end of the backplane.
Also it serves as an extension to the backplane, since each module has a connector on its front panel serving as a 1:1 connection to the backplane. This way the slot doesn't get wasted for nearly no function.
In addition to that, each supply voltage has a dedicated LED to show if the voltage is available or if a fuse has blown.
And that is all there is to it.

6.2. Host Bus Controller

The Host Bus Controller (HBC) is the heart of the ModLab. It acts as master for the backplane busses as well as user interface. Since it is quite a busy module, I'll try my best to explain each function seperately

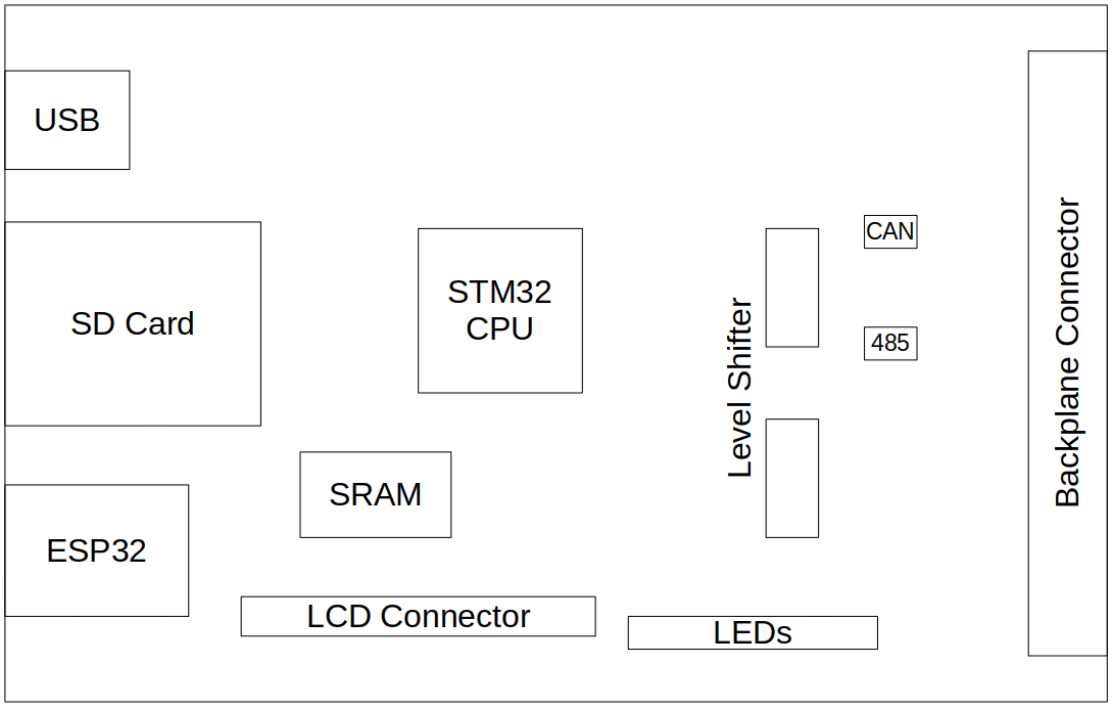

Figure 6.1.: PCB Overview HBC

6.2.1. CPU

The core of the HBC is the STM32F767 microcontroller. It controls everything in the entire system and provides the user interfaces.
With 2MB of Flash and 512kB RAM it has more than plenty of storage both for the program code as well as process data and variables. Also at 216MHz max clock rate it is really fast and can process everything in a timely manner. At least that was the thought behind choosing this chip.

6.2.2. Additional Hardware

Also it has a lot of interfaces which are used on the PCB. Additionally to the interfaces mentioned in chapter 5.4, internally the module has two SPI busses, one for the touch controller and one for the ESP32 coprocessor. A USB interface for debugging and monitoring is inlcuded as well.
For logging and config storage I included an SD card slot.
The FSMC (Flexible SRAM Memory Controller) interface of the microcontroller is connected to an external 512k x 16 SRAM chip. I thought about including an external flash memory for application software, but decided against it, since the internal flash seemed more than enough. The external SRAM might come in handy tho if I decide to build modules like logic analyzers which have a ton of data coming in at once.
The ESP32 module will provide a webinterface for monitoring and diagnosis, but not for controlling the system. This decision was made from a security stand point.
For Interfacing with the whole system, a 5” Nextion TFT LCD is connected via UART.
How the protocols on the interfaces will look like, will be discussed in another chapter.

6.3. Diode Tester

The Diode Tester is actually quite simple in design. It consists of 4 constant current sources which are tuned to about 2.2mA, 5mA, 10mA and 20mA respectively. Due to this I can enable up to 16 different currents through the Device Under Test (DUT) and even make a crude characteristic curve out of it. Following is a simplified schematic of the current sources.

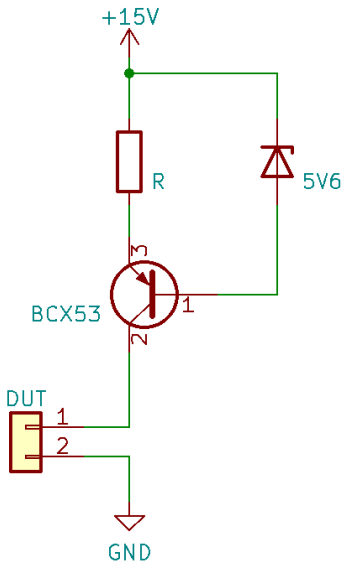

Figure 6.2.: Current Source Schematic

The current is configured via the resistor according to Kirchhoff's loop rule. $\mathrm{U_{be}}$ of the transistor is typically 0.7V, which means the voltage across the resistor is fixed at $5.6\mathrm{V} - 0.7\mathrm{V} = 4.9\mathrm{V}$. With the values of 2.2k, 1k, 499 and two 499 in parallel respectively, the following currents are available:

$$\frac{4.9\mathrm{V}}{2.2\mathrm{k}\Omega} = 2.22\mathrm{mA}$$
$$\frac{4.9\mathrm{V}}{1\mathrm{k}\Omega} = 4.9\mathrm{mA}$$
$$\frac{4.9\mathrm{V}}{499\Omega} = 9.81\mathrm{mA}$$
$$\frac{4.9\mathrm{V}}{249.5\Omega} = 19.63\mathrm{mA}$$

In V1 I tried to enable the sources via a P-channel MOSFET, which for some strange reason didn't work. So I decided for V2 to use small reed relais I had in storage.

The module also has an STM32F103 controller on board which is connected to the backplane via CAN and I2C. I did both since I hadn't yet decided which interface I am going to use for the module.

6.3.1. Parameters

No.	Size	Description
10h	2 Byte	DUT Voltage at 2 mA
11h	2 Byte	DUT Voltage at 5 mA
12h	2 Byte	DUT Voltage at 7 mA
13h	2 Byte	DUT Voltage at 10 mA
14h	2 Byte	DUT Voltage at 12 mA
15h	2 Byte	DUT Voltage at 15 mA
16h	2 Byte	DUT Voltage at 17 mA
17h	2 Byte	DUT Voltage at 20 mA
18h	2 Byte	DUT Voltage at 22 mA
19h	2 Byte	DUT Voltage at 25 mA
1Ah	2 Byte	DUT Voltage at 27 mA
1Bh	2 Byte	DUT Voltage at 30 mA
1Ch	2 Byte	DUT Voltage at 32 mA
1Dh	2 Byte	DUT Voltage at 35 mA
1Eh	2 Byte	DUT Voltage at 37 mA
20h	1 Byte	Starting Current
21h	1 Byte	Ending Current

Table 6.1.: Diode Tester Parameter List

6.3.2. Operation Modes

Start Measurement with “Enable Output” Command.

Mode	Description
0	Single Value Measurement Starting Current only
1	Coarse Characteristic Curve Starting Current to Ending Current Only Hardware Sources enabled
2	Medium Characteristic Curve Starting Current to Ending Current Every other possible current
3	Fine Characteristic Curve Starting Current to Ending Current Every possible current

Table 6.2.: Diode Tester Parameter List

6.3.3. Channels/Outputs

Enabled Channels:

- ADC0

6.4. Waveform Generator DDS

For most of the design I have to thank “Daumemo” from daumemo.com. I wanted to use the AD9833 waveform generator IC and his guide helped me design the circuit around it.
The board mainly consists of the STM32F103 controller, the AD9833 signal generator, a VCA (voltage controlled amplifier) and an output amplifier. Gain and Offset are controlled via PWM, the Interface to the AD9833 is SPI.
The Signal Generator is capable of generating Sine, Triangle and Square waves from 0.1Hz up to 12.5MHz with a 0.1Hz resolution.

I will add more info once I get to program the module.

6.4.1. Parameters

6.4.2. Operation Modes

6.4.3. Channels/Outputs

6.5. Symmetric Linear Power Supply (SymPSU)

The design for the symmetric power supply is derived from the user “The Big One” over on Hackaday. I adopted it to my STM32 controller and to 3V3 supply voltage. Also I switched the control voltage generation from a dedicated DAC to the PWM-channels of the STM32. I am using the internal ADC on the STM32 as well.
With these design changes, the SymPSU can provide the following voltages:

- Positive voltage
  - 0 to 12V @ 0 to 1.2A
- Negative voltage
  - 0 to -12V @ 0 to 1.2A

After tweaking the code, I programmed the following resolutions:

- Voltages
  - 10 mV / step
- Currents
  - 0.1 mA / step

I am quite happy with these specs, but will likely tweak the design in the future a bit. I still have some issues that are currently fixed in software, rather than in hardware.

6.5.1. Parameters

No.	Size	Description
10h	2 Byte	Set Positive Voltage [mV]
11h	2 Byte	Set Positive Current [mA]
12h	2 Byte	Set Negative Voltage [mV]
13h	2 Byte	Set Negative Current [mA]
20h	2 Byte	Measured Positive Voltage [mV]
21h	2 Byte	Measured Positive Current [mA]
22h	2 Byte	Measured Negative Voltage [mV]
23h	2 Byte	Measured Negative Current [mA]

Table 6.3.: SymPSU Parameter List

6.5.2. Operation Modes

n/a.

6.5.3. Channels/Outputs

Enabled Channels:

- ADC0
- ADC1
- ADC2
- ADC3
- PWM0
- PWM1
- PWM2
- PWM3
- Output Relais

6.6. Switching Mode Power Supply

6.6.1. Parameters

6.6.2. Operation Modes

6.6.3. Channels/Outputs

7. Interface Protocols

As the system is quite big and has a lot of interfaces, here will be the place to discuss all these.
Mainly the USB-Interface, SPI-Interface for the ESP32-Communication and CAN-Interface are of interest. But also the secondary I2C and general SPI interfaces will be defined here.

7.1. USB Monitor

TBD

7.2. CAN

7.2.1. Overview

The CAN bus is the main communication bus for the ModLab. The HBC is sequentially talking to all the connected modules and sending commands to them if needed. This includes regular status updates as well as configurations. The CAN bus itself is divided into seperate layers, which will be discussed now.

1. **Physical Layer**
   The physical layer of the CAN interface consists of the CAN module on the STM32 controllers as well as a SN65VHD232 CAN Transceiver. CAN Rx and TX are sent on the same differential interface, making the bus half-duplex.

   In order to get enough data around in a feasable time, the Interface will be clocked at 1MHz speed.

2. **Transport Layer**
   The transport layer descibes the way the data will be sent. Since standard CAN frames can only handle 8 bytes at once, I thought about implementing something here, to increase the data transfer.

   After careful thinking tho, I decided agains a special transport protocol like ISO-TP, because the overhead creates more delays than I am comfortable with. I just have to be careful designing the command structure to not include commands with too many arguments.

3. **Application Layer**
   Here is, where it gets interesting. After we can now send plenty of data, I need to define the actual transmission protocol of the payload for the modules. Since CAN is designed as such that every node reads the data at the same time, I have to implement a procedure to let the modules decide if the received message is of interest for them or not.

   This can be achieved with two different methods. One is by encoding the module address in the ID section of the module, the other is sending the module in question as first byte in the payload.

   I decided to implement the first option to save on transmission time. The ID field in the datagrams will be used both for encoding the module address, as well as indicaton for special cases like warnings and errors in my protocol. Also with this method I can selectively talk to each module without blocking processing time for checking if the message is really meant for them.

   To clarify: if I talk about address, I mean the module address. I will refer to the ID as a whole (address + Message Type) as ID.

7.2.2. General Structure

The general structure of the protocol uses a ping—pong approach. The command value sent to the module will always be returned to the sender as well to clarify that this is the reply tho this command. Even tho I might sacrifice some bandwidth with this method, I think it will be worthwhile in future implementations.
Because of this, the structure is always “Command — Value” in both directions.

7.2.3. ID Endcoding

The ID structure will be used as specified in the following table. The “xx” in the ID will be placeholders for the module address.

ID	Type
7xxh	Auto Message (informational)
6xxh	Reply to Command (ACK)
5xxh	Standard Command
4xxh	Warnings / State Changes
3xxh	Reply to Command (NACK)
2xxh	Processing Error
1xxh	Critical Error
0xxh	reserved

Table 7.1.: CAN-Messages

7.2.4. Auto-Messages

TBD

7.2.5. Base-Addresses

Base Address	Module
10h	HBC
20h	Diode Tester
30h	Symmetrical Linear Power supply
34h	reserved
38h	reserved
3Ch	24V Switching Mode Power Supply
40h	DDS Waveform Generator

Table 7.2.: CAN Addresses

7.2.6. Commands

Overview:

Value	Command
01h	Reset Module
05h	Ping / Discovery
10h	Set 8bit Parameter
11h	Get 8bit Parameter
14h	Set 16bit Parameter
15h	Get 16bit Parameter
20h	Get Status Information
40h	Enable Output
41h	Disable Output
60h	Set Operation Mode (TBD)
61h	Get Operation Mode (TBD)

Table 7.3.: CAN-Commands

7.2.6.1. 01h — Reset Module

Force Reset of Module.
Command:

Value	Length	Description
01h	1 Byte	Reset Module Command

Reply:

Value	Length	Description
01h	1 Byte	Reset Module Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions

7.2.6.2. 05h — Ping / Discovery

Ping Module to just get an answer.
Command:

Value	Length	Description
05h	1 Byte	Ping Command

Reply:

Value	Length	Description
05h	1 Byte	Ping Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions

7.2.6.3. 10h — Set 8bit Parameter

Write 8bit value to module parameter.
Command:

Value	Length	Description
10h	1 Byte	Set 8bit Parameter Command
Index	1 Byte	Parameter Index, see Module Description
Value	1 Byte	Value to be written

Reply:

Value	Length	Description
10h	1 Byte	Set 8bit Parameter Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions

7.2.6.4. 11h — Get 8bit Parameter

Read 8bit value from module parameter.
Command:

Value	Length	Description
11h	1 Byte	Get 8bit Parameter Command
Index	1 Byte	Parameter Index, see Module Description

Reply:

Value	Length	Description
11h	1 Byte	Get 8bit Parameter Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions
Value	1 Byte	Value to be read; 0 if NACK

7.2.6.5. 14h — Set 16bit Parameter

Write 16bit value to module parameter.
Command:

Value	Length	Description
14h	1 Byte	Set 16bit Parameter Command
Index	1 Byte	Parameter Index, see Module Description
Value	2 Byte	Value to be written

Reply:

Value	Length	Description
14h	1 Byte	Set 16bit Parameter Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions

7.2.6.6. 15h — Get 16bit Parameter

Read 16bit value from module parameter.
Command:

Value	Length	Description
15h	1 Byte	Get 16bit Parameter Command
Index	1 Byte	Parameter Index, see Module Description

Reply:

Value	Length	Description
15h	1 Byte	Get 16bit Parameter Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions
Value	2 Byte	Value to be read; 0 if NACK

7.2.6.7. 20h — Get Status Information

Get status information from module.
Command:

Value	Length	Description
20h	1 Byte	Get Status Information Command

Reply:

Value	Length	Description
20h	1 Byte	Get 16bit Parameter Command
Status	1 Byte	Error-Status see Error Descriptions
Channel Info	1 Byte	bit 0..3: PWM-Channel 0..3 bit 4..5: ADC-Channel 0..3 bits are 0 if channel not implemented

7.2.6.8. 40h — Enable Output

Enable the Module Output.
Command:

Value	Length	Description
40h	1 Byte	Enable Output Command

Reply:

Value	Length	Description
40h	1 Byte	Enable Output Command
Status	1 Byte	Error-Status see Error Descriptions

7.2.6.9. 41h — Disable Output

Disable the Module Output.
Command:

Value	Length	Description
41h	1 Byte	Disable Output Command

Reply:

Value	Length	Description
41h	1 Byte	Disable Output Command
Status	1 Byte	Error-Status see Error Descriptions

7.2.6.10. 60h — Set Operation Mode

Set Module into Operation Mode x.
Command:

Value	Length	Description
60h	1 Byte	Set Operation Mode Command
Mode	1 Byte	Operation Mode, see Module Description

Reply:

Value	Length	Description
60h	1 Byte	Set Operation Mode Command
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions

7.2.6.11. 61h — Get Operation Mode

Get Module Operation Mode.
Command:

Value	Length	Description
61h	1 Byte	Get Operation Mode Command

Reply:

Value	Length	Description
61h	1 Byte	Get Operation Mode Command
Mode	1 Byte	Operation Mode, see Module Description
Status	1 Byte	Error-Status in case of NACK Otherwise set to 0 see Error Descriptions

7.2.7. Error Descriptions

7.2.7.1. Status Byte

Description of the Status Byte in Reply Messages. If a Bit is set to 0, it is inactive. A 1 indicates an active error.
The lower Nibble indicates protocol errors, a mismatch condition can mean both “not available” and “wrong length”.
The upper Nibble indicates internal errors of the module.

Bit	Description
0	Index Mismatch
1	Data Mismatch
2	Channel Mismatch
3	Operation Mode Mismatch
4	Processing Error
5	I/O Error
6	Unknown Command
7	reserved

7.3. TFT Display

The Communication Interface with the TFT display is a 5V TTL UART with 250.000 baud of standard 8N1 protocol.

7.3.1. HBC to TFT

For sending commands or instructions to the TFT display, everything needs to be in ASCII followed by 3 Bytes of 0xFF.
Nearly all commands directly set properties of the elements placed on screen. For example setting the value of a floating point number (here: Element name x1) with two decimal places to the value “3.30” would be as follows:
“x1.val=330”
Note, that no decimal point is set. The display can only handle integer numbers. Decimal points are handled by the corresponding number box element on screen.
Here is an overview of the most important commands for Interfacing. All examples are made with textbox t1 and float number x1.

Command	Description
t1.txt=“foo”	set text
t1.aph=127	set opacity to fully visible
x1.val=330	set numeric value
vis t1,0	set textbox invisible. a 1 at the end makes it visible again

7.3.2. Change Device Page

So far, every module class has its own page on the display. So only these modules need to be actively monitored.
If the user changes the page, the following command will be sentby the TFT:
“0x55 0x##”where ## is the following number:

Number	Description
00h	Home Screen / General Settings
01h	Diode Tester
02h	Function Generator
03h	Symmetrical Power Supply
04h	Switching Mode Power Supply

7.3.3. Set Input Process Data

Depending on the module, this command can have a variable length of data trailing it. The general instruction across all modules begins with “0x5C”.

length	Description
1 byte	Module ID
1 byte	Operation mode
1 byte	Start Current Index
1 Byte	Stop Current Index

Table 7.4.: Data Structure Diode Tester

length	Description
1 byte	Module ID
4 byte	Frequency - LSB first
2 byte	Amplitude - LSB first
2 byte	Offset - LSB first

Table 7.5.: Data Structure Function Generator

length	Description
1 byte	Module ID
2 byte	Positive Voltage - LSB first
2 byte	Positive Current - LSB first
2 byte	Negative Voltage - LSB first
2 byte	Negative Current - LSB first

Table 7.6.: Data Structure Symmetric Power Supply

7.3.4. Set Output

The Set Output instruction only exists for modules that actually have a disconnectable output. The command generally follows the format “0x50 0xIS”, where I is the module sub-ID (numbered 1-4) and S is the state of the output (0 = off, 1 = on).

7.3.5. General Settings (Home Screen)

General settings are mostly for debug purposes, but can be activated and deactivated on the home screen. The instruction begins with “0x5A” and is followed by the data below:

Data	Description
10h	Disable CAN-Log over USB
11h	Enable CAN-Log over USB
20h	Disable TFT-Log over USB
21h	Enable TFT-Log over USB
30h	Don’t simulate missing modules
31h	Simulate all missing modules

7.4. ESP SPI

TBD

7.5. Backplane SPI

TBD

7.5.1. 5V SPI

TBD

7.5.2. 3V3 SPI

TBD

7.6. I2C

TBD

7.6.1. Standard Mode

TBD

7.6.2. Fast Mode

TBD

Addenda