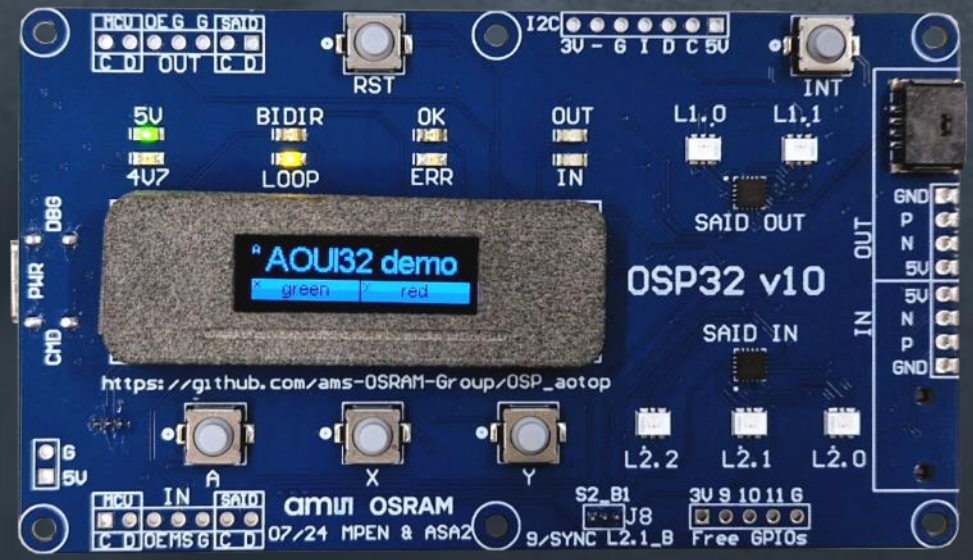
Sense the power of light

OSP on Arduino – Training

Using the *Arduino OSP evaluation kit*



2025 May 8

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

Part 6 – Middleware (topo)

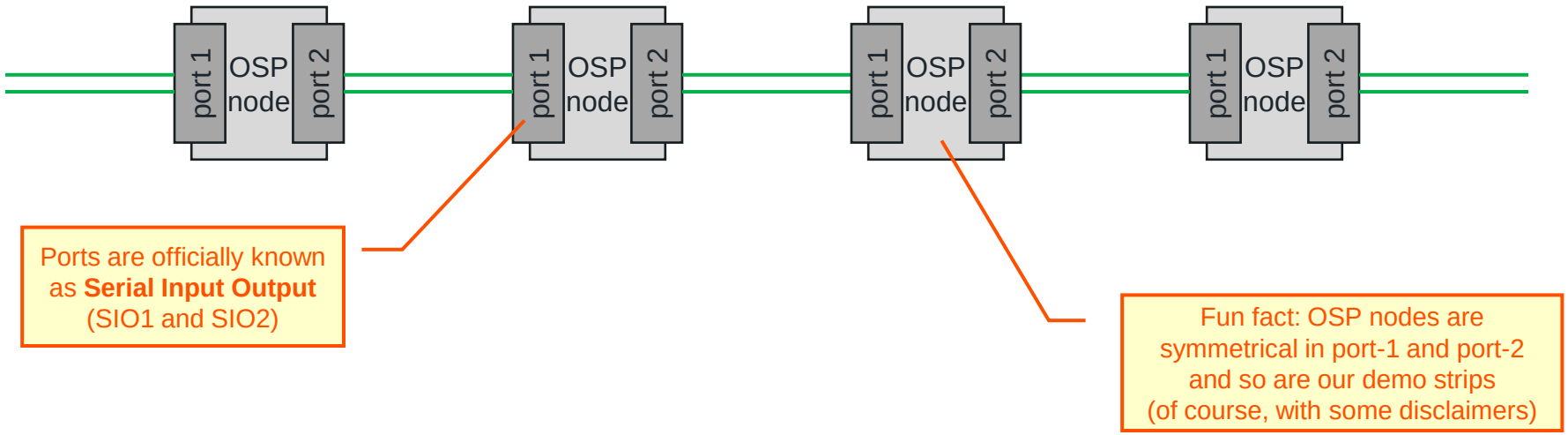
Part 7 – Command interpreter

Part 8 – Miscellaneous

Open System Protocol (OSP)

Daisy chain and ports

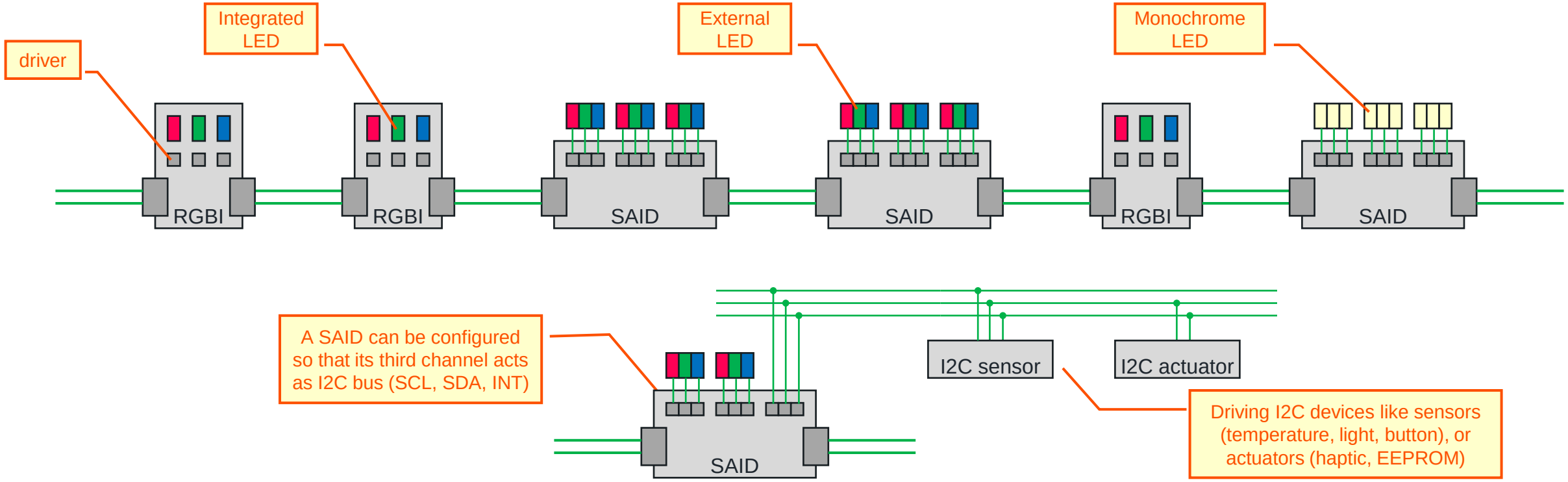
- Chips that are OSP compliant can be daisy chained – using two wires (differential signaling for automotive “LVDS”)
- They forward “byte trains” with commands, formatted following the OSP rules – known as telegrams



Open System Protocol (OSP)

Drivers and LEDs

- OSP nodes typically have LED drivers integrated
- Some OSP nodes have red/green/blue LEDs integrated – example is RGBI (E3731i)
- Others allow connecting external LEDs (eg side-lookers or monochrome) – example is SAID (AS1163)

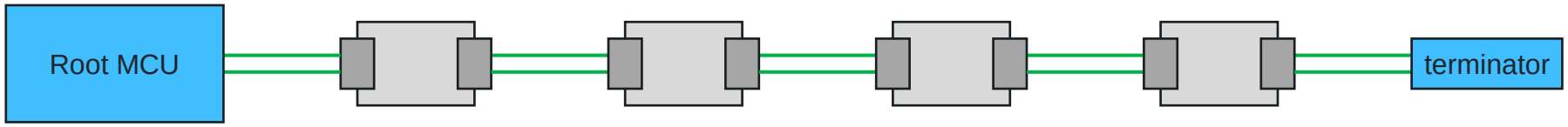


P1 prereq

Open System Protocol (OSP)

Handling both ends of the daisy chain

- At the root of the OSP chain there is a Root MCU which initiates the telegrams
- At the end of the OSP chain there is a terminator: with that the last node *knows* that it is the last one

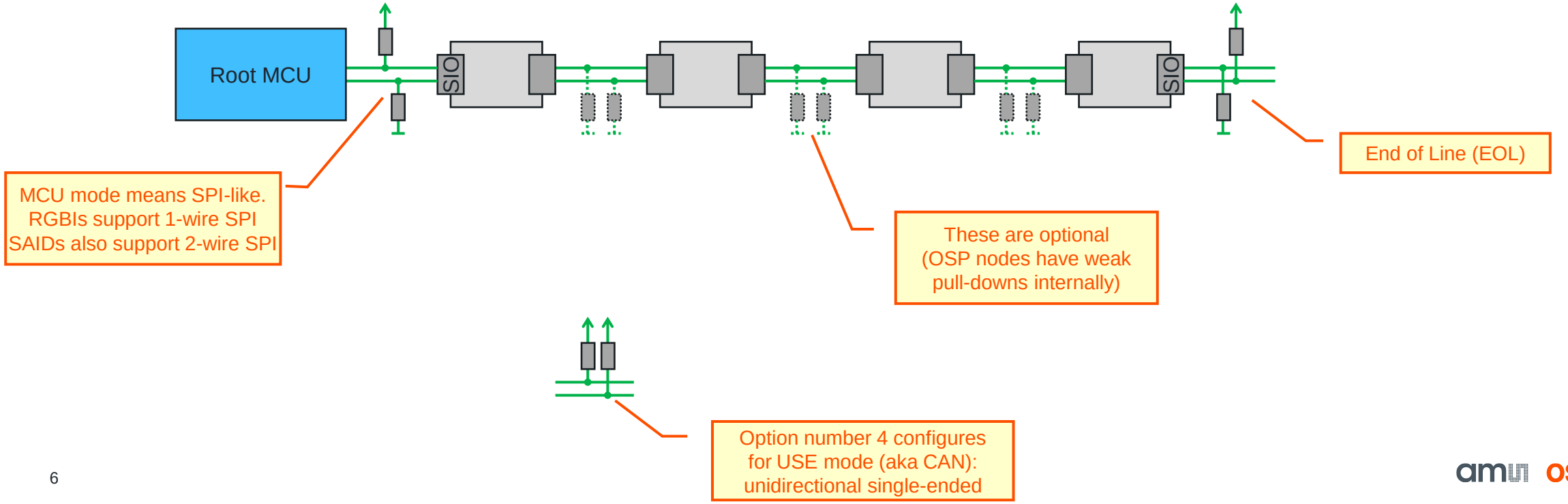


ams OSRAM

Open System Protocol (OSP)

Advanced: SIO configuration

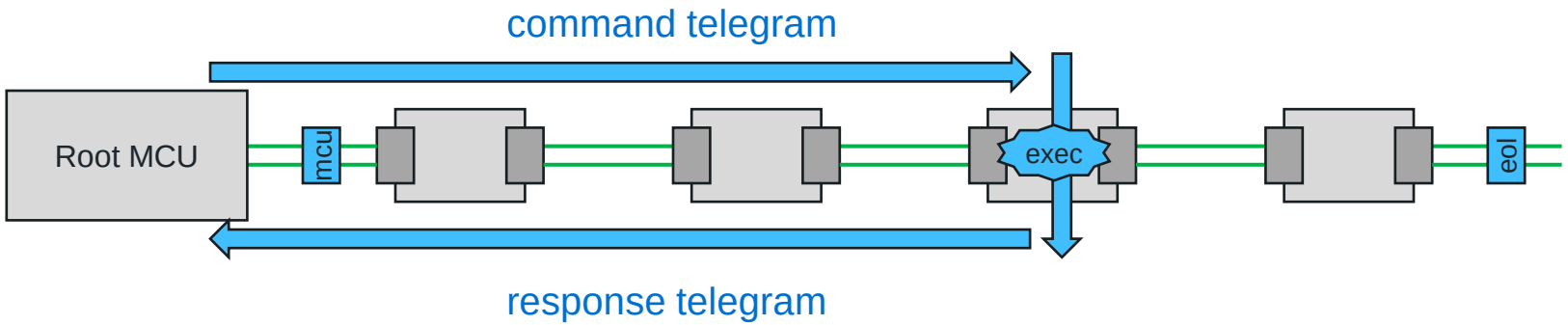
- The incoming SIO port of the first OSP node needs to be configured for MCU mode (so that it talks SPI instead of LVDS)
- The outgoing SIO port of the last OSP node needs to be configured for EOL mode (this is the terminator)
- All other ports are implicitly configured for LVDS (low voltage differential signaling)



Open System Protocol (OSP)

Response telegrams in bidirectional communication (BiDir)

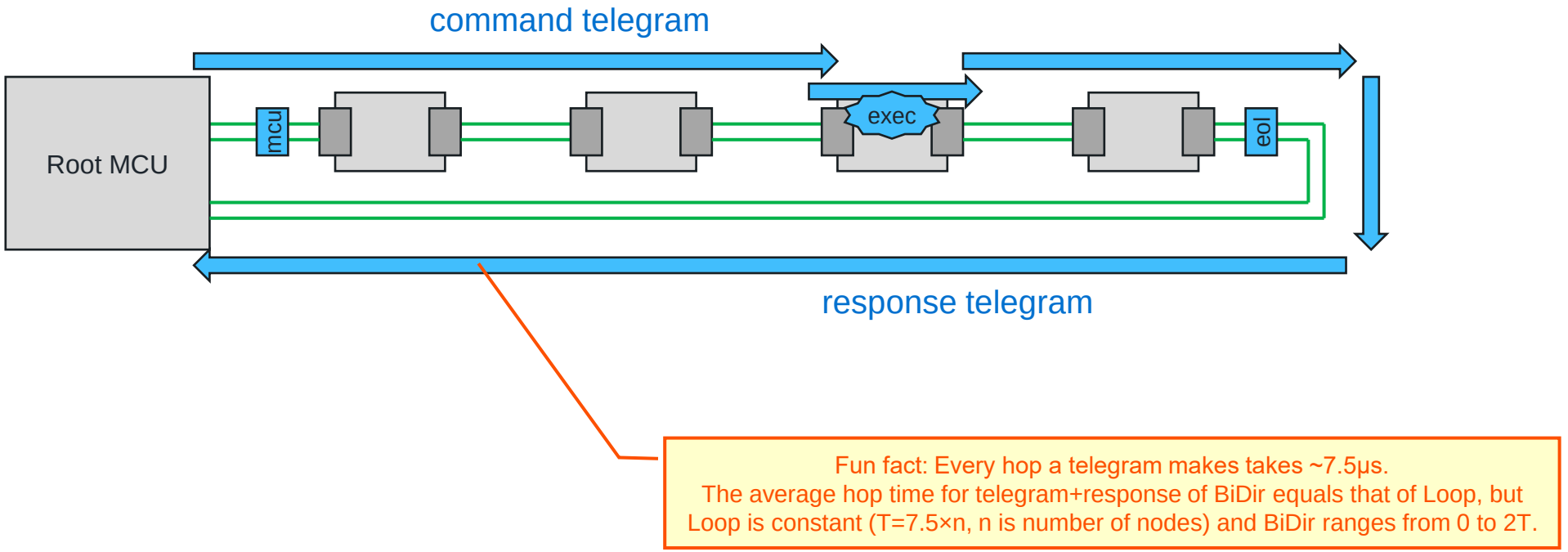
- Some command telegrams cause the addressed node to reply with a response telegram
- Response telegrams “return” to the MCU, this is called bidirectional communication (“BiDir”)



Open System Protocol (OSP)

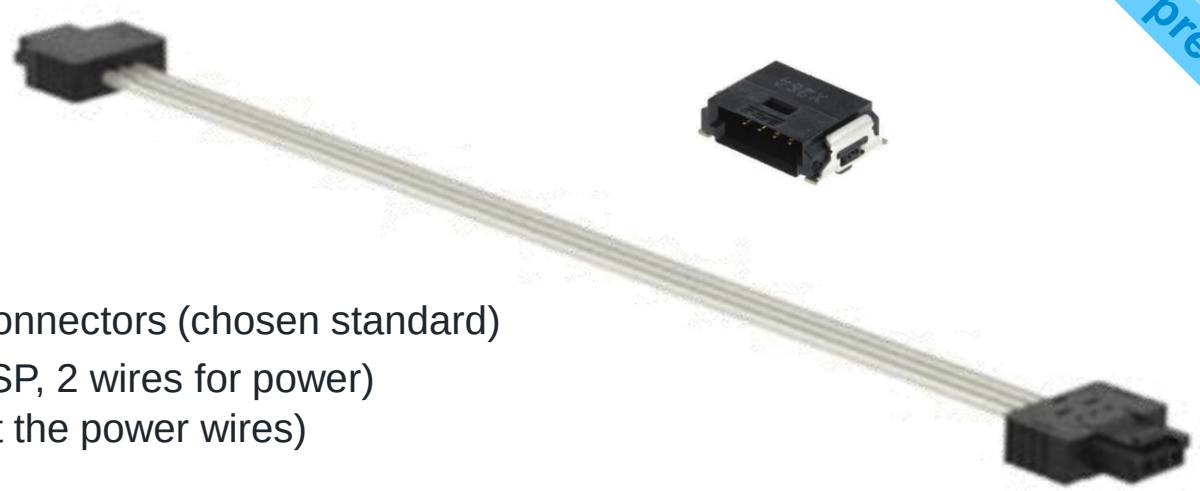
Response telegrams in unidirectional communication (Loop)

- Response telegrams can also move forward; that is called unidirectional communication (“Loop” mode)
- This does require the MCU to have two unidirectional ports (instead of one bidirectional)
- (a telegram configures a node to BiDir/return or Loop/forward mode)

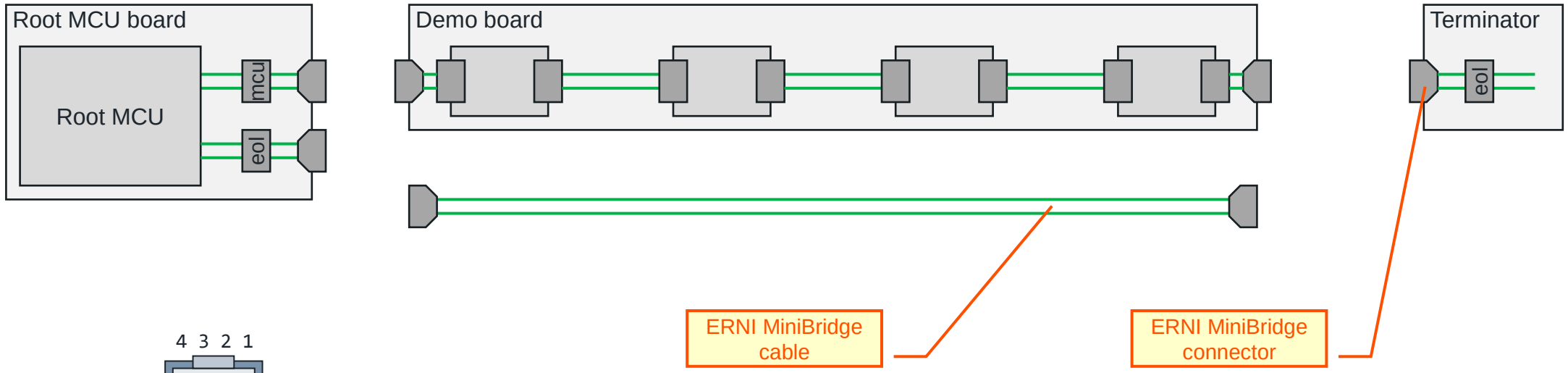


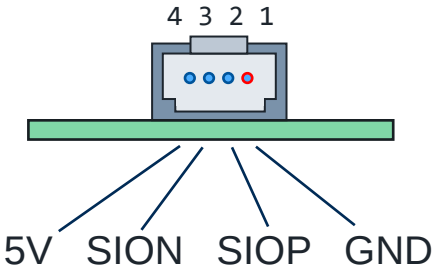
Open System Protocol (OSP)

Components in an evaluation kit



- All eval kit boards expose their ports using ERNI MiniBridge connectors (chosen standard)
- The eval kit uses 4-wire connectors and cables (2 wires for OSP, 2 wires for power)
  (the diagrams only show the 2 OSP wires – in green – and not the power wires)



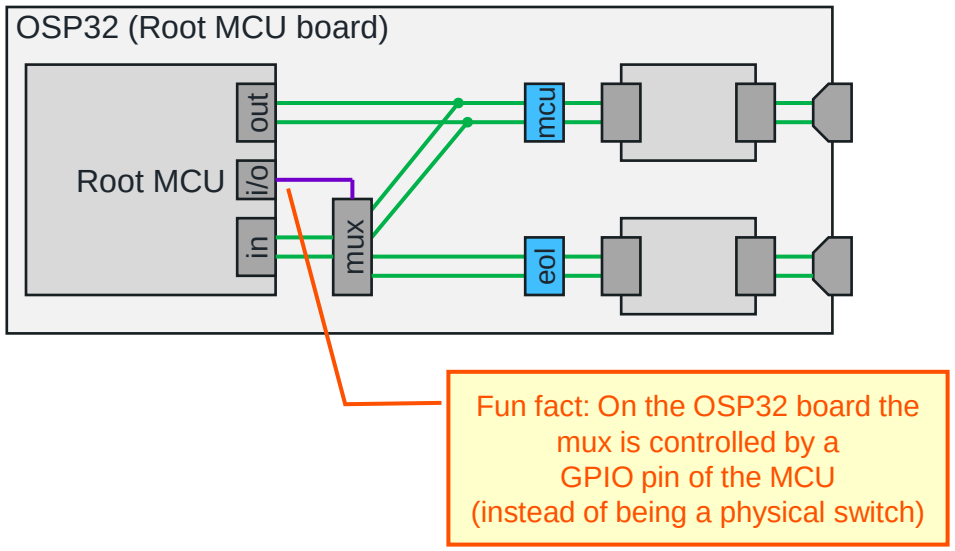


https://www.distrelec.nl/en/ribbon-cable-27mm-cores-200mm-black-erni-839017/p/14362654

Open System Protocol (OSP)

Flexibility in an evaluation kit

- The MCU typically uses one SPI block for transmission (“out”) and a second SPI block for reception (“in”)
- The MCU board in an evaluation kit has a switch/mux to allow evaluation of BiDir as well as Loop
- (production boards make a choice for either one – hardwired)



Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

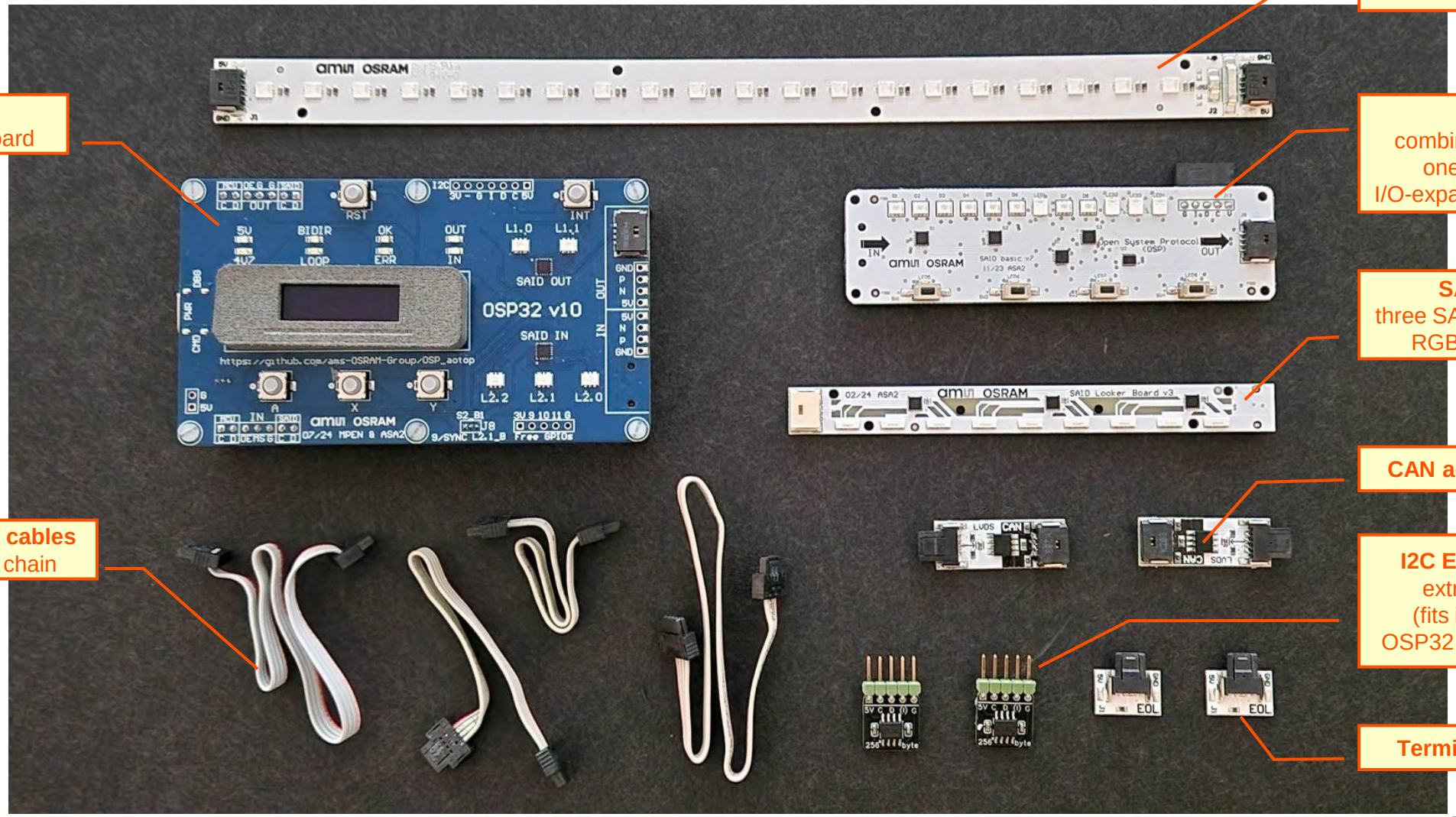
Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

Arduino OSP evaluation kit

Kit contents



RGBIstrip
20 RGBIs

OSP32
Root MCU board

SAIDbasic
combines SAIDs and RGBIs
one SAID has I2C with
I/O-expander, EEPROM, header

SAIDlooker
three SAIDs, with each 3
RGB side lookers

CAN adapters

ERNI MiniBridge cables
to form an OSP chain

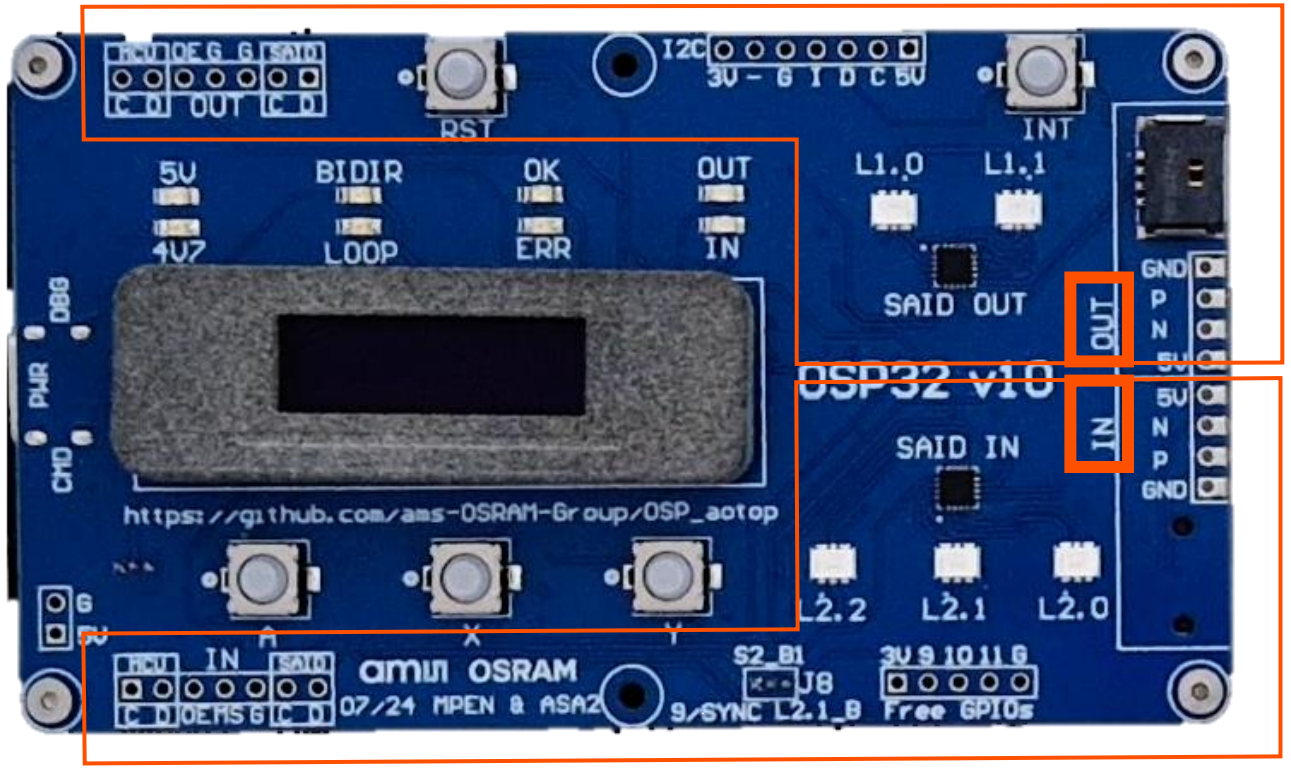
I2C EEPROM stick
extra EEPROM
(fits in headers of
OSP32 and SAIDbasic)

Terminators

Arduino OSP evaluation kit

The OSP32 board – OUT and IN

Components dedicated to **transmitting** telegrams to the start of the OSP chain; the “OUT” part



Components dedicated to **receiving** telegrams from the end of the OSP chain; the “IN” part

Arduino OSP evaluation kit

The OSP32 board – nodes

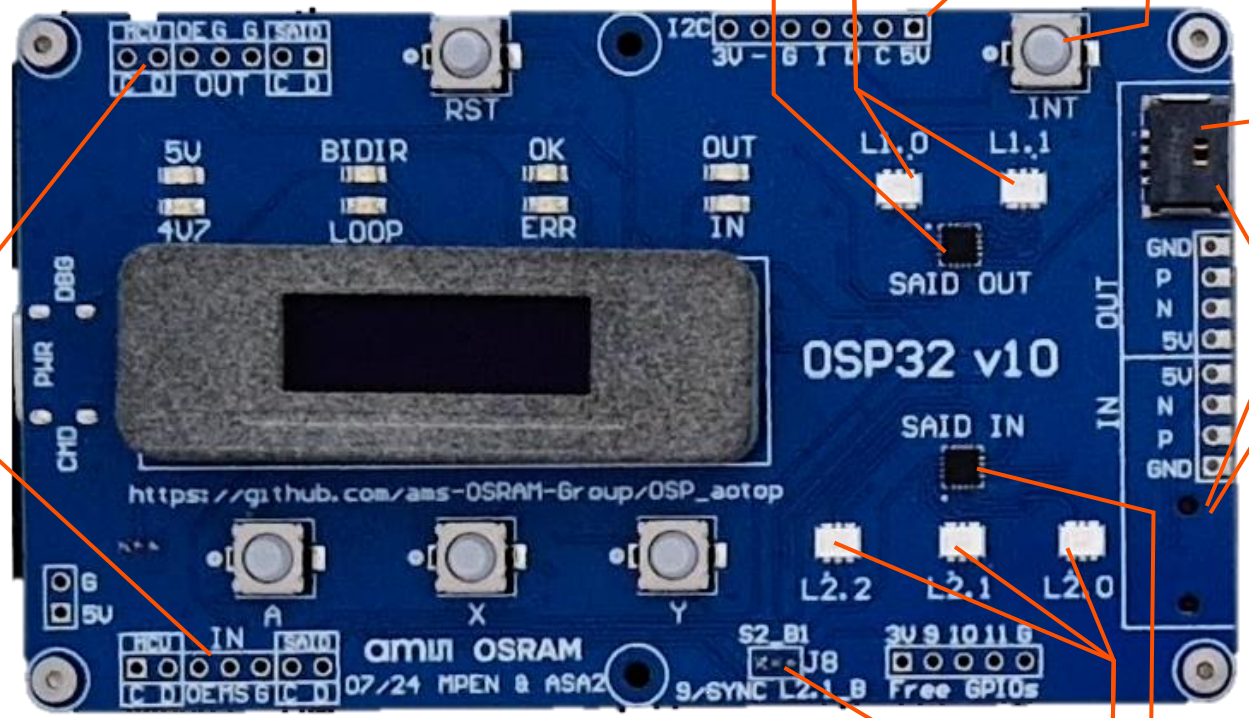
Root MCU ESP32S3
(at the bottom)

SAID OUT
(MCU mode towards ESP)

Two RGBs
(L1.0 and L1.1)

Third channel I2C
EEPROM and header

INT button
also part of I2C



Upstream connectors on top of the board, downstream connectors at the bottom

ERNI connector OUT

ERNI connector IN

Trace pins for OUT
(logic analyzer)

Trace pins for IN
(logic analyzer)

Three USB plugs
(at the bottom)

SAID IN
(EOL mode towards ESP)

Three RGBs
(L2.0, L2.1, L2.2 reversed)

Jumper for SYNC of SAID IN
(special use case)

"CMD" powers via ESP (weak) but allows flashing/serial (**preferred**)
"DBG" powers via ESP (weak) allows JTAG debugging
"PWR" powers directly (no voltage drop) – for stand alone usage

Arduino OSP evaluation kit

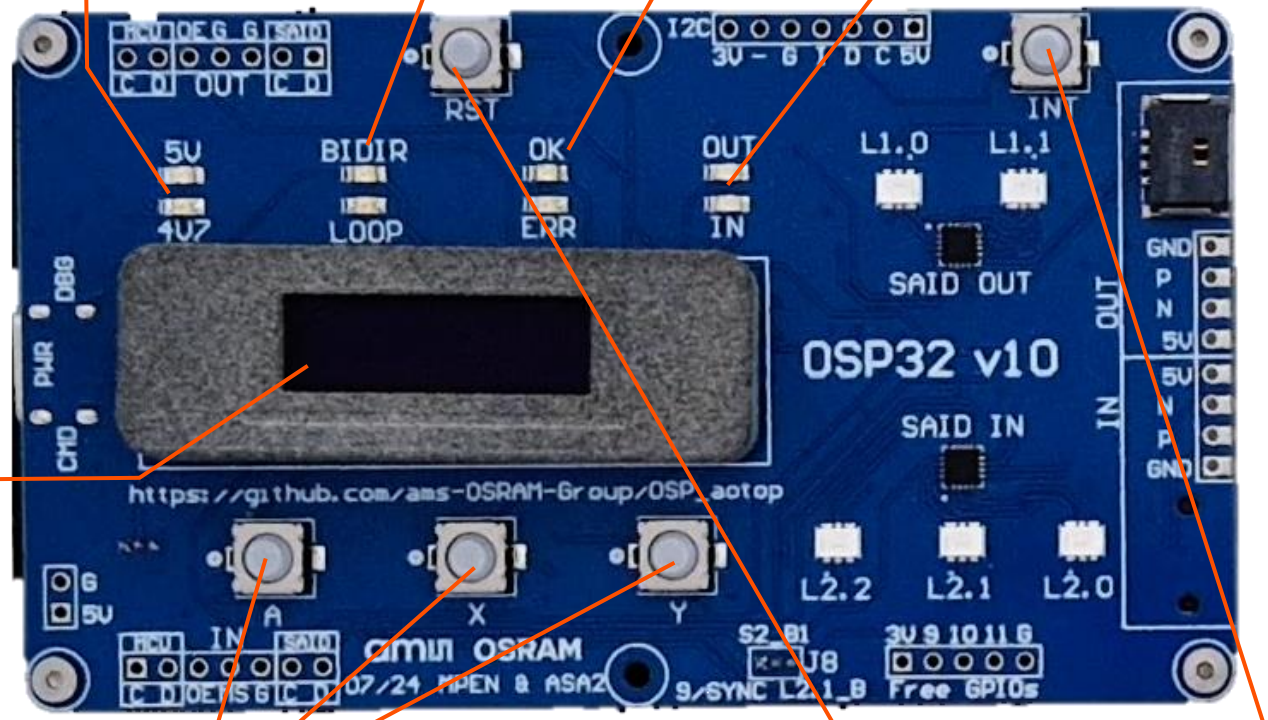
The OSP32 board – UI features

Voltage monitor
(goes orange when voltage drops below 4V7)

IN/OUT mux indicator
(flags BiDir or Loop)

Red/green signaling LEDs
(under control of ESP: OK or ERR)

IN & OUT traffic indicator
(blinks for each telegram)



128×32 OLED
(under control of ESP)

Three buttons (A/X/Y)
(under control of ESP: A for next app)

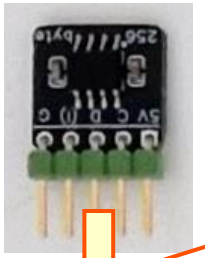
ESP reset button
(does not reset OLED or OSP nodes)

INT button
also part of I2C

Arduino OSP evaluation kit

The SAIDbasic board – one of the demo boards

SAIDs 3rd channel is I2C
(for IOX, EEPROM, header)



I2C EEPROM stick
(optionally inserted)

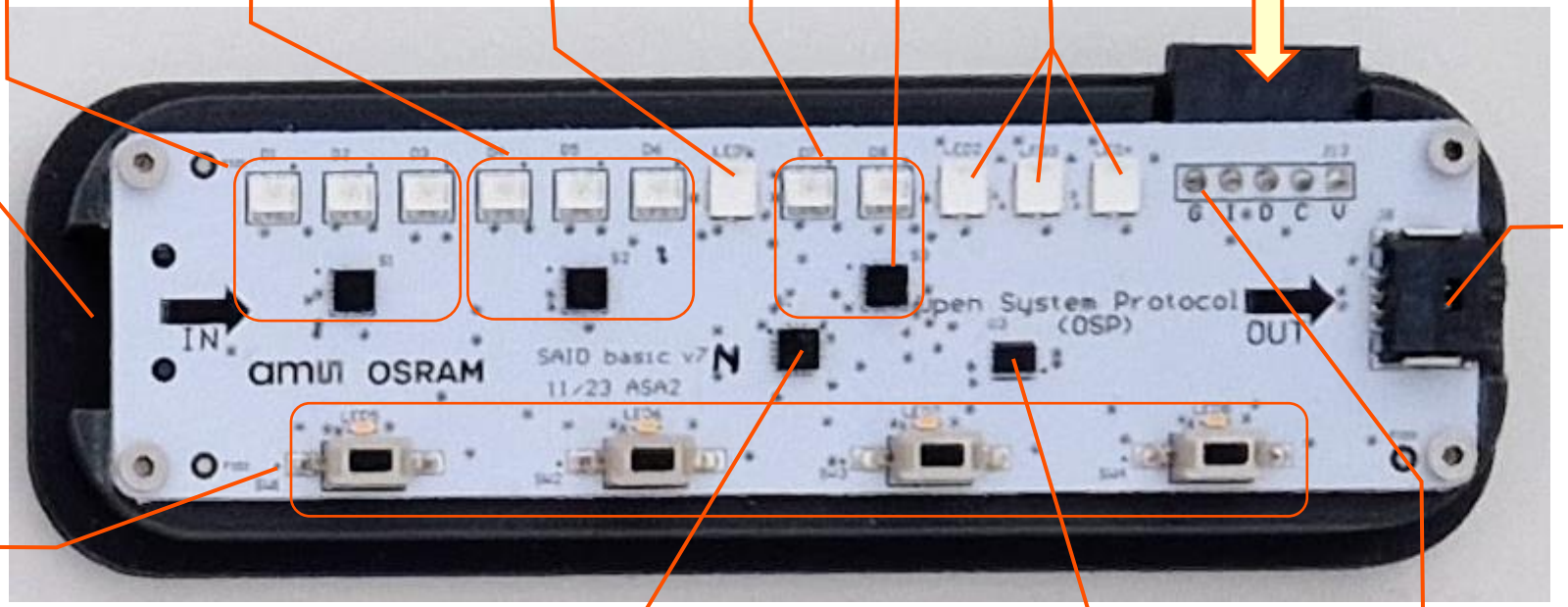
Node 1: SAID
(with 3 RGBs)

Node 2: SAID
(with 3 RGBs)

Node 3: RGBI
(is 1 RGB)

Node 4: SAID
(2 RGBs and I2C)

Node 5, 6, 7: RGBI
(each 1 RGB)



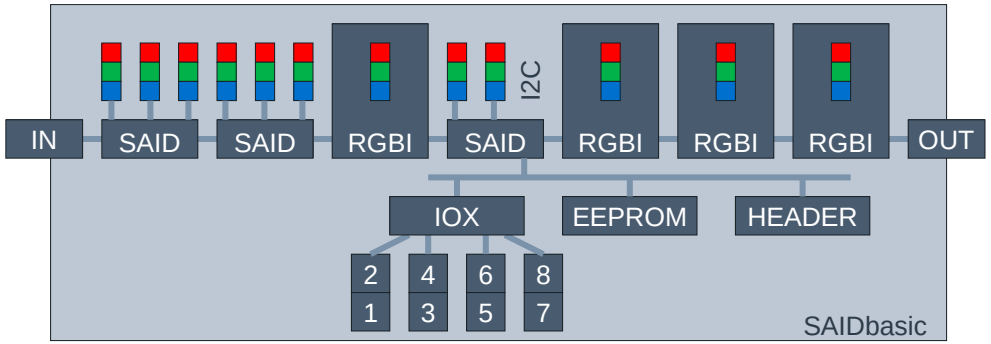
IN connector
(at bottom)

downstream connectors at the bottom of the board

OUT connector
(at top)

upstream connectors at the top of the board

User interface
(buttons and indicators connected to I/O-expander)



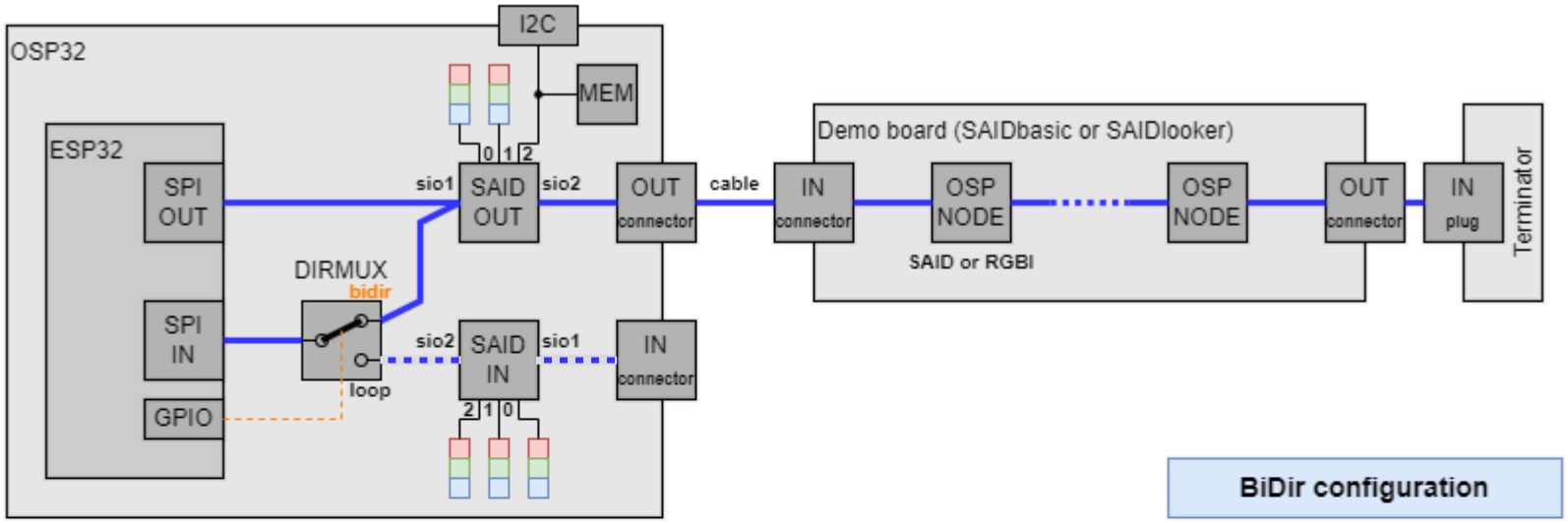
I2C I/O-expander
(with 4 buttons and 4 LEDs)

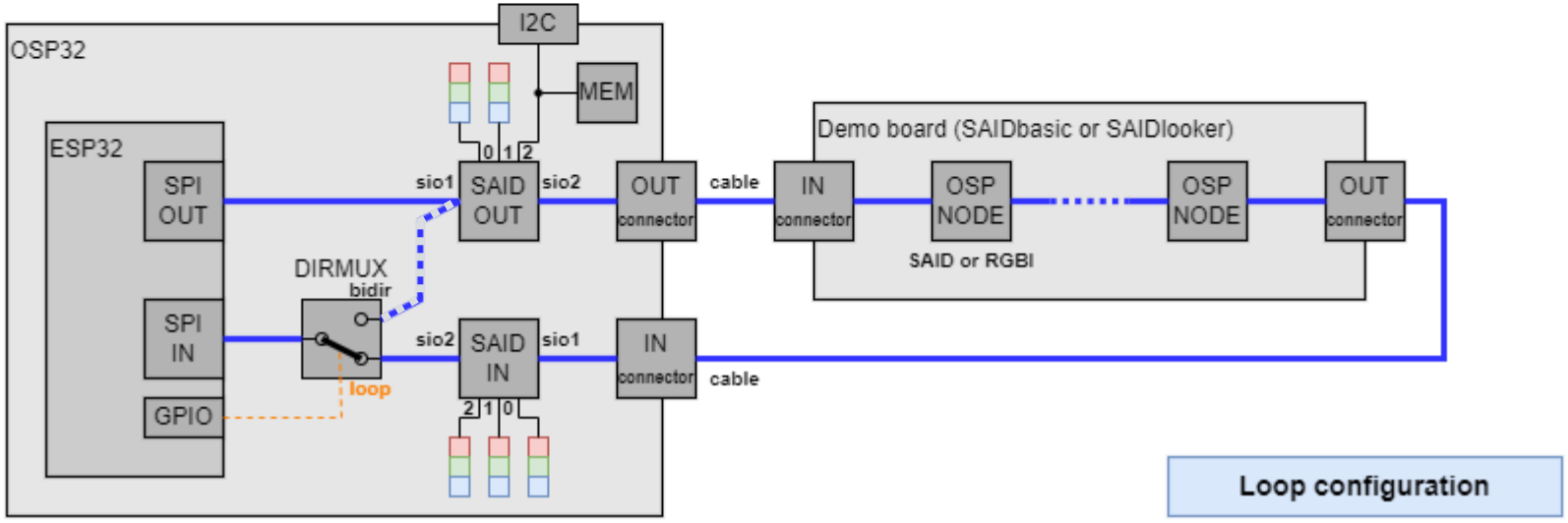
I2C EEPROM
(memory of 256 bytes)

I2C header
(SCL/SDA/INT and GND/5V)

Arduino OSP evaluation kit

The OSP32 board – high level schematics





ams OSRAM

Intermezzo - debugging

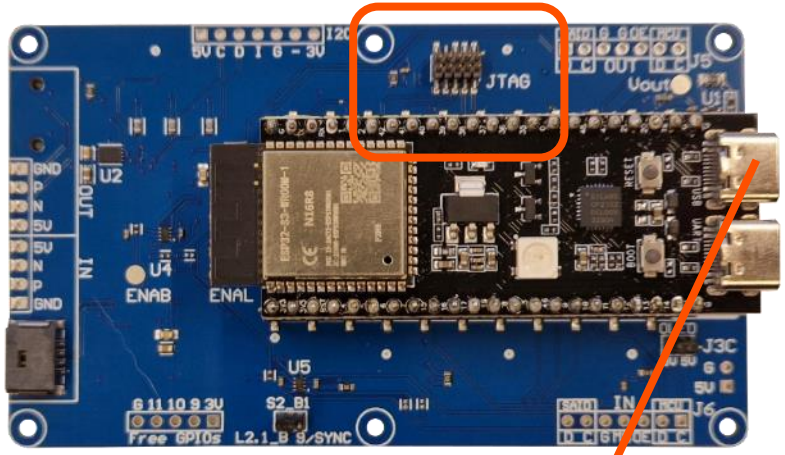
OSP32, ESP32S3 and Arduino

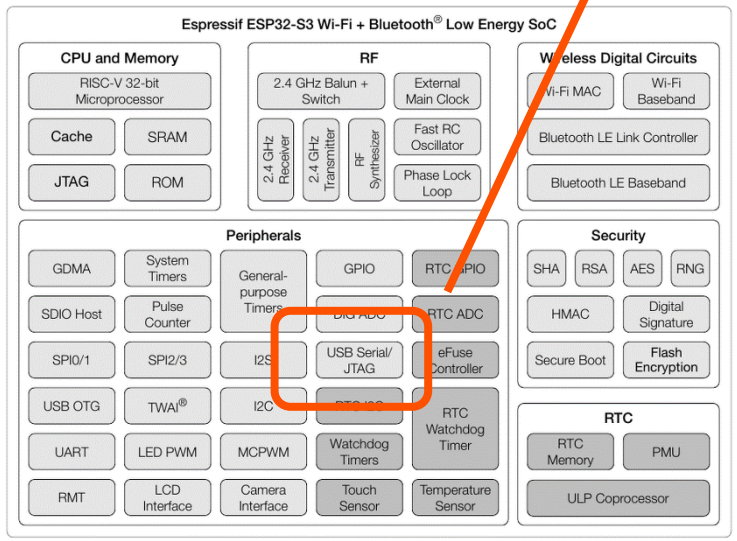
- The ESP32S3 has a JTAG port that can be used for debugging
- It is pinned-out on the OSP32 board
- You need to connect a JTAG-USB debug probe (eg Segger J-link) to it

- Better yet, the ESP32S3 has a JTAG probe built-in
- Connecting a USB cable to the USB port labeled DBG is enough
- This is one of the reasons for selecting the ESP32**S3**

- In both cases the IDE needs to support debugging
- The Arduino IDE 2.x has support for debugging ESP32S3 (this is the chip on the “Nano ESP32” board from Arduino)

- Arduino broke debugger support for ESP32 – but is fixed now
- The Getting Started document has a Debugging section at the end explaining how to use an Arduino IDE for debugging.





Homework

To be completed *before* the training

Make sure all software is downloaded, installed and running.

Follow the chapter “Installation” at https://github.com/ams-OSRAM/OSP_aotop/blob/main/gettingstarted.md

Steps

- Install Arduino IDE
- Via Arduino board manager add ESP32 boards
- Via Arduino library manager add the OSP libraries
- Compile and run example aoosp_min

Hope you did that

- Because needed for the first hands-on of this training
- … which is now … 😱

Assignment – Training0 - saidbasic

Get familiar with the evaluation kit

User manual of this demo in OSP_aotop\extras\manuals\saidbasic.pptx

0

In Arduino IDE:
- From library *aotop* open the example *saidbasic*
- Build using “ESP32S3 Dev Module”
- Flash/upload it to the ESP (use USB “CMD”)
- Connect the SAIDbasic board in BiDir mode
- Don’t forget the terminator

1

In app *Animation script*
- Change animation by plugging in I2C EEPROM stick
- FPS up and down with X and Y button

2

In app *Running LED* (press A switch for next app)
- Add second demo board (SAIDlooker) to see autoconfig
- Dim up and down with X and Y button

3

In app *Switch flag*
- Change flags (press one of 4 buttons, see indicator LED)
- Dim up and down with X and Y button

Use the demo

4

In app *Dithering*
- Use mobile phone video recording with high frame rate
- Toggle dithering

5

In Arduino IDE (with app Running LEDs active)
- Open Serial monitor on correct port (and baud 115200)
- Type **`topo dim`**, then **`topo dim 1`**, and **`topo dim 1024`**
- Press A button and see message on terminal

Meet the command interpreter

6

Signaling LEDs on OSP32 board
- Green OK heartbeat and BiDir LED is on
- Remove terminator and reset
- Notice red ERR (io heartbeat) and message on OLED
- Connect Loop wire and reset
- Notice heartbeat and Loop LED is on
- Press Y for high FPS in *Animation script*: blue OUT on
- Start *Switch flag*; blue IN on (reading buttons)
- Switch USB cable from CMD to PWR; check 5V/4V7 LEDs

Understand the signaling LEDs

7

- Attach the RGBi strip and flip it
- Attach the SAIDlooker strip and flip it

Intermezzo – Power Architecture

5V0 vs 4V7

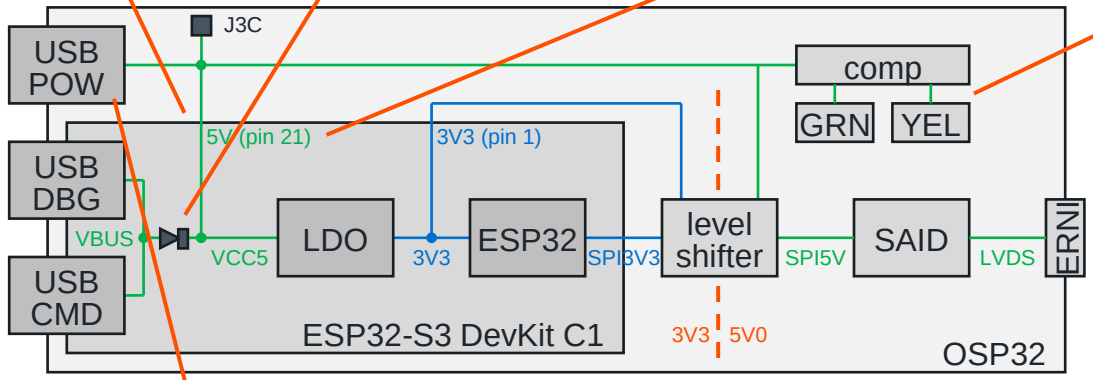
USB plugs
For ESP flashing, printf, and debugging

Pin 21 power supply *input*
when DBG/CMD is not used
external power to LDO (from POW/J3C)

Protection diode
higher voltage on J3C does not reach PC via DBG/CMD

Pin 21 power supply *output*
when DBG/CMD is plugged and POW/J3C not
VCC5 net is then 4V7 due to protection diode

Voltage indicator
the comparator lights green when 5V0 and yellow when 4V7



USB DBG, USB CMD
USB plugs for power and data

USB POW
Power plug in USB form factor; carries power but no data

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

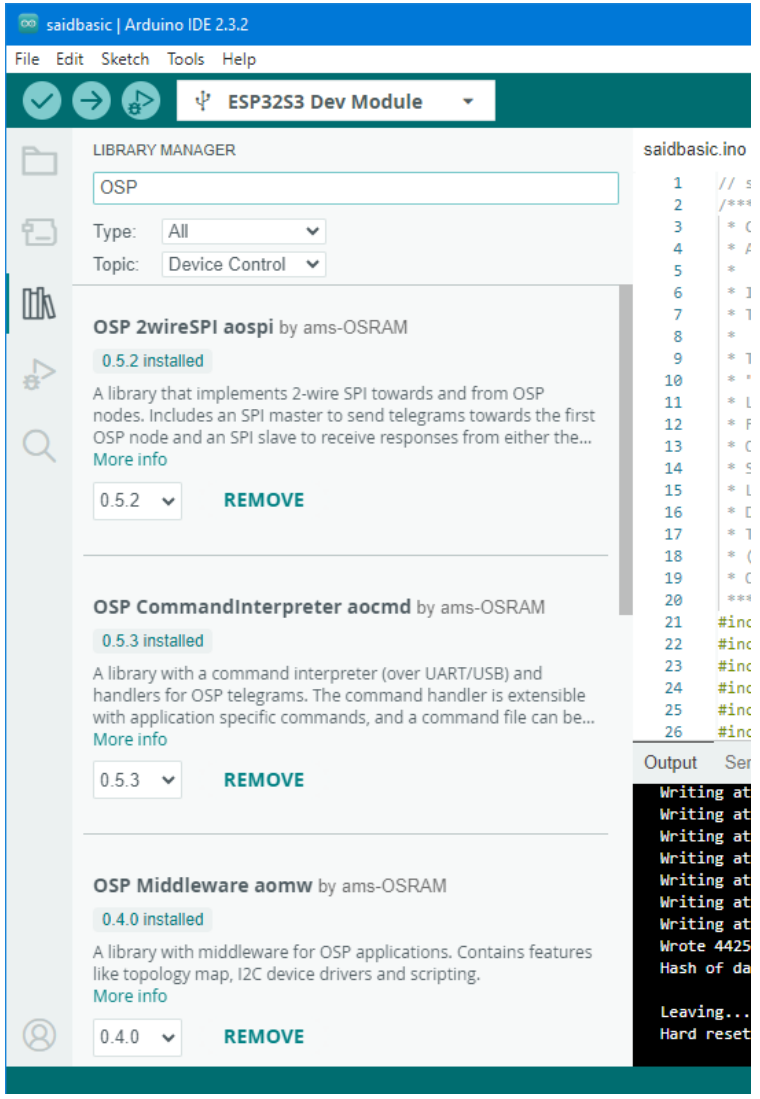
Software – the *oalibs*

Where to find it

- Software is written to support OSP protocol, with SAID and RGBI
- Software targets ESP32S3 (fast, memory rich, cost effective, no favor auto MCU)
- Software written for Arduino (known, well documented, lots of support, free)
- Software is written in the form of Arduino libraries
  - eight in total, nicknamed “aolibs” (Arduino OSP from ams OSRAM)
  - contain source code (reusable library code)
  - and many example sketches – “sketch” is Arduino name for (source of) executable
  - including sketches of the official demo
  - and various sources of documentation

- Published on GitHub (list is here)
- Each library has its own repo
- One library is the entry point https://github.com/ams-OSRAM/OSP_aotop
- Helpful for quick browsing and jumping around (links)

- Software is also registered at Arduino
- Can be found and installed within Arduino IDE (using the library manager)
- Library *aotop* “pulls in” all others (dependency list)



Libraries overview

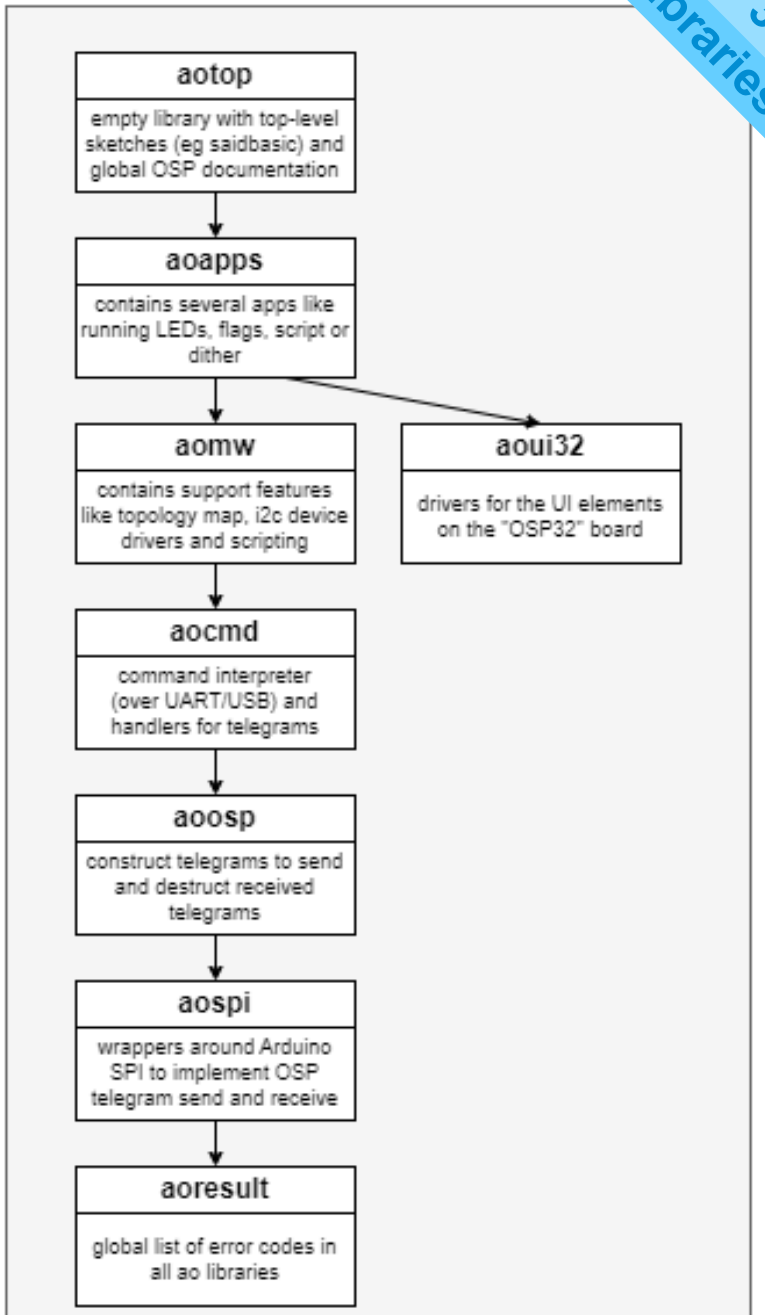
Hierarchy

- The software has been split in eight libraries
- They depend on each other in a mostly linear fashion

- Each library has three names
  - Full: **`OSP ToplevelSketches aotop`** (the way Arduino calls it)
  - Repo: **`OSP_aotop`** (the way GitHub calls it)
  - Short: **`aotop`** (the way we call it)

- The library at the top of the hierarchy is, rightly named, *aotop*

Coding rule
If a library's short name is **`aolll`**, then a module (c/h) of that library has filename **`aolll_mmm`** and an external symbol will be called **`aolll_mmm_sss`**



Arduino OSP libraries from ams-OSRAM

Libraries – 1

OSP ResultCodes aoresult

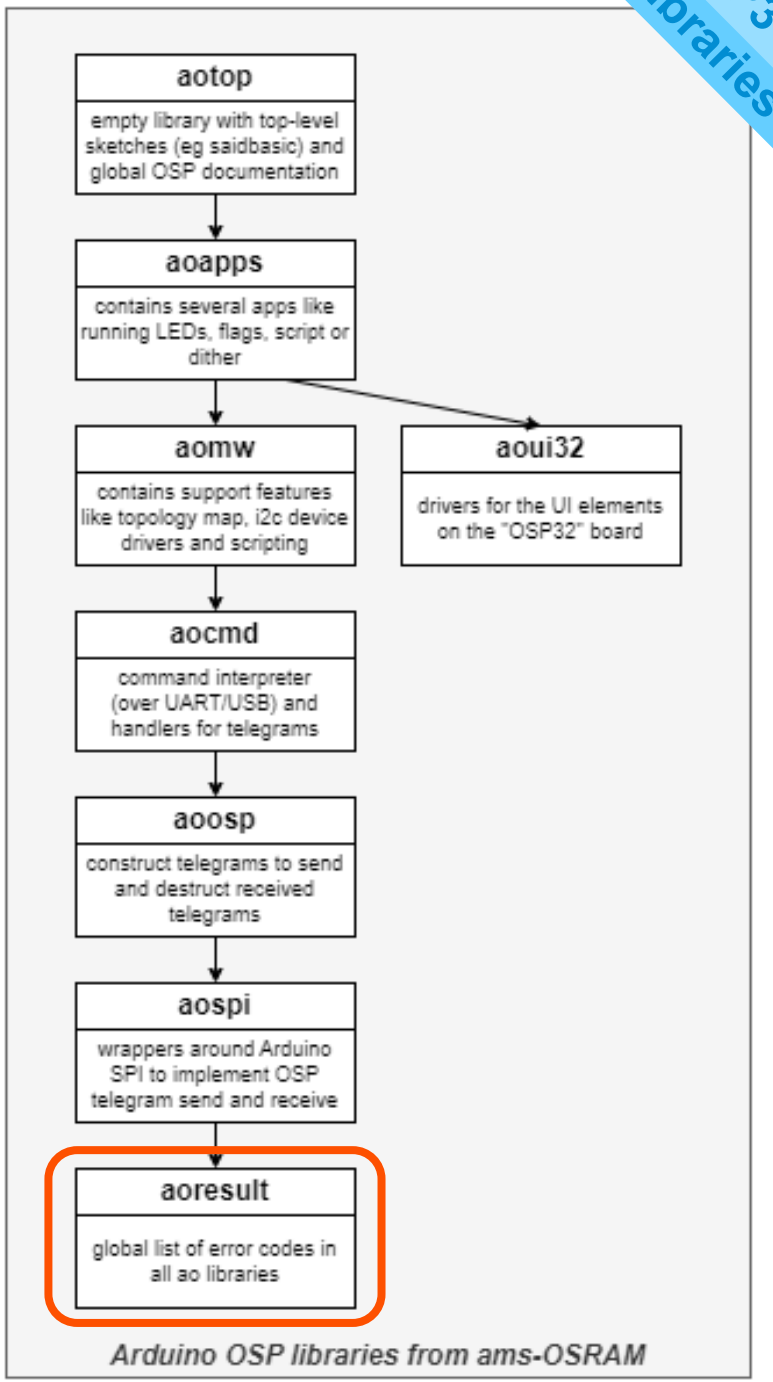
- This library is rather empty

- It contains the global list of error codes `aoresult_t`
  - next library aospi needs error codes
  - but aospi might need to be replaced with another library (eg different MCU)
  - therefore, aospi could not be owner of the error codes
  - a library “below” was needed for them

- Contains a function to map and error code to a readable string.

- It also contains a definition of “assert”

```
typedef enum aoresult_e {
  aoresult_ok              =  0,
  ...
  aoresult_dev_i2cnack     = 43,
} aoresult_t;

const char * aoresult_to_str(aoresult_t result);
```



Libraries – 2

OSP 2wireSPI aospi

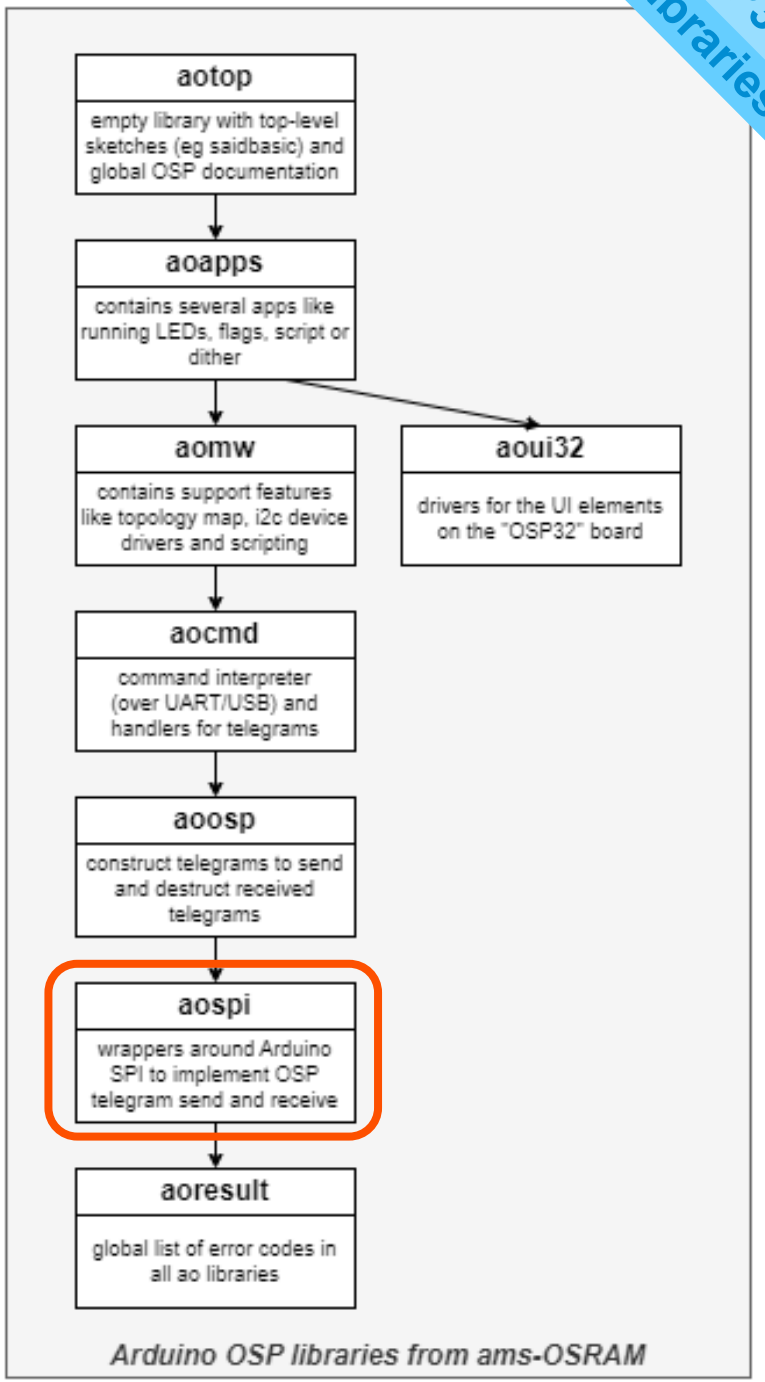
- This library implements 2-wire SPI communication
  - 2-wire SPI is a new feature of SAID
  - it is faster than 1-wire SPI (also in SAID, only one in RGBI)
- This lib sends telegrams to an OSP chain and receives responses from it
- Sending and receiving is on the level of byte arrays (“buffers”).

- The caller is responsible to ensure the buffer contains the correct details
  - Header (preamble, destination address, payload size, telegram ID)
  - Telegram parameters
  - Matching CRC
  - All bits packed according to the OSP standard.

- The OSP32 board has a mux to chose between “Loop" and “BiDir".
- This library also has functions to control that mux.

```
aoresult_t aospi_tx(const uint8_t * tx, int txsize);
void aospi_dirmux_set_loop();
void aospi_dirmux_set_bidir();
```

If you use a different protocol (e.g. 1-wire SPI), a different MCU (e.g. NXP S32K144 instead of ESP32S3) or less flexibility (e.g. no BiDir/Loop with auto select), this library would be replaced.



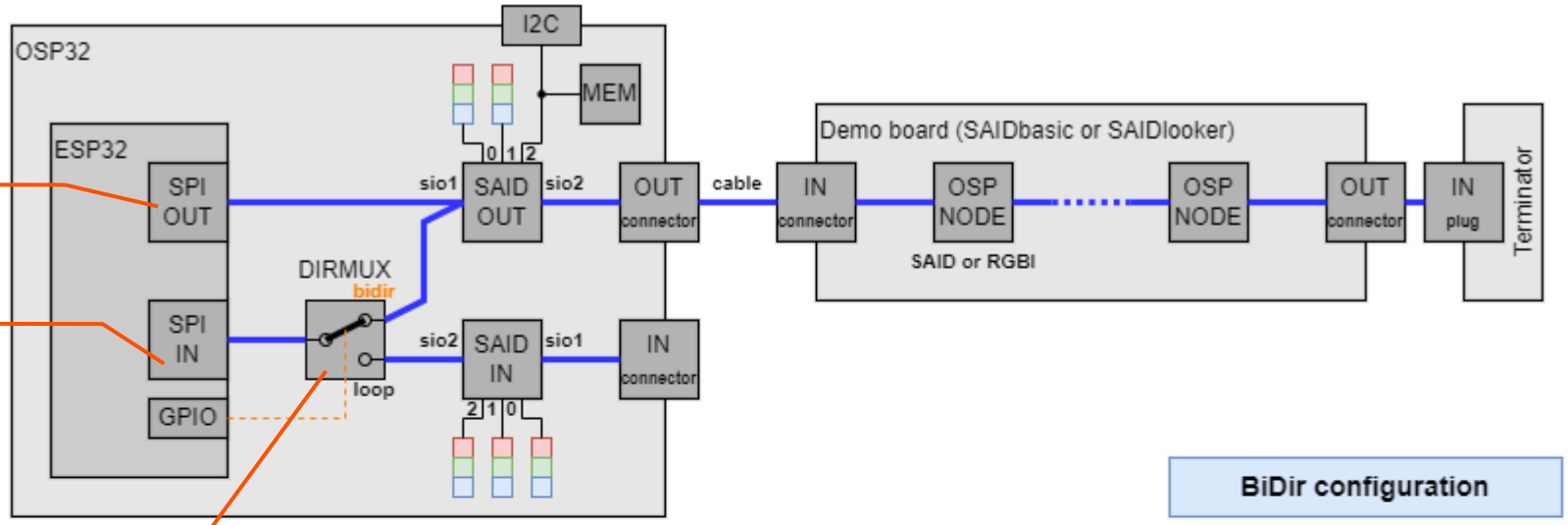
Libraries – 2 – Recap OSP32 board design

OSP 2wireSPI aospi

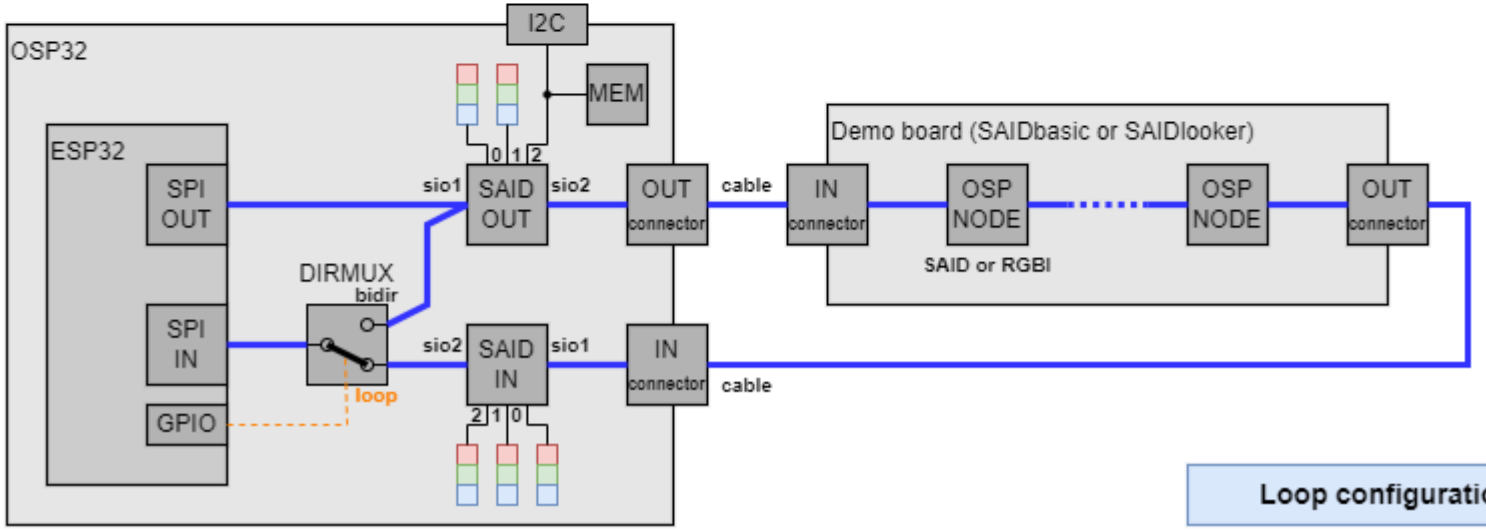
aospi implements SPI OUT (master)

… and the SPI IN (slave)

… and mux control



BiDir configuration



Loop configuration

ams OSRAM

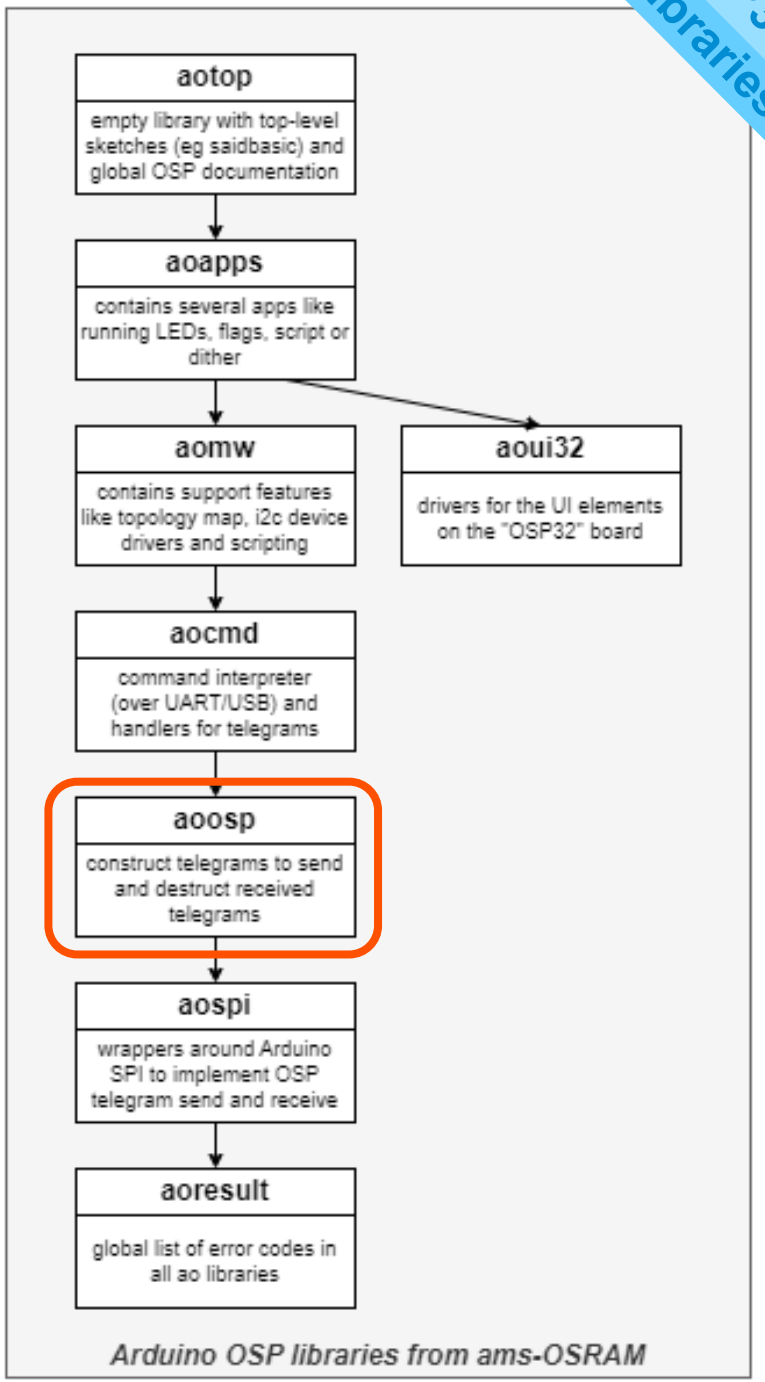
Libraries – 3

OSP Telegrams aoosp

- This library implements the OSP communication at telegram level
- The API has one function per telegram
  - with as argument the node's address and the telegram parameters
  - and optionally out parameters from the response telegram
- The library does the packing of byte buffers (and computing CRC)
- and unpacking from byte buffers for responses (and checking CRC)
- and finally calls *aospi* to transmit the buffer

- Any application for OSP is expected to use the *aoosp* library
- and thus *aospi* (or substitute) and *aoresult*.

```
aoresult_t aoosp_send_reset(uint16_t addr );
aoresult_t aoosp_send_gosleep(uint16_t addr );
aoresult_t aoosp_send_goactive(uint16_t addr );

// with parameters (other then destination)
aoresult_t aoosp_send_setmult(uint16_t addr, uint16_t groups);

// with response
aoresult_t aoosp_send_readstat(uint16_t addr, uint8_t * stat);
```



Arduino OSP libraries from ams-OSRAM

Libraries – 4

OSP CommandInterpreter aocmd

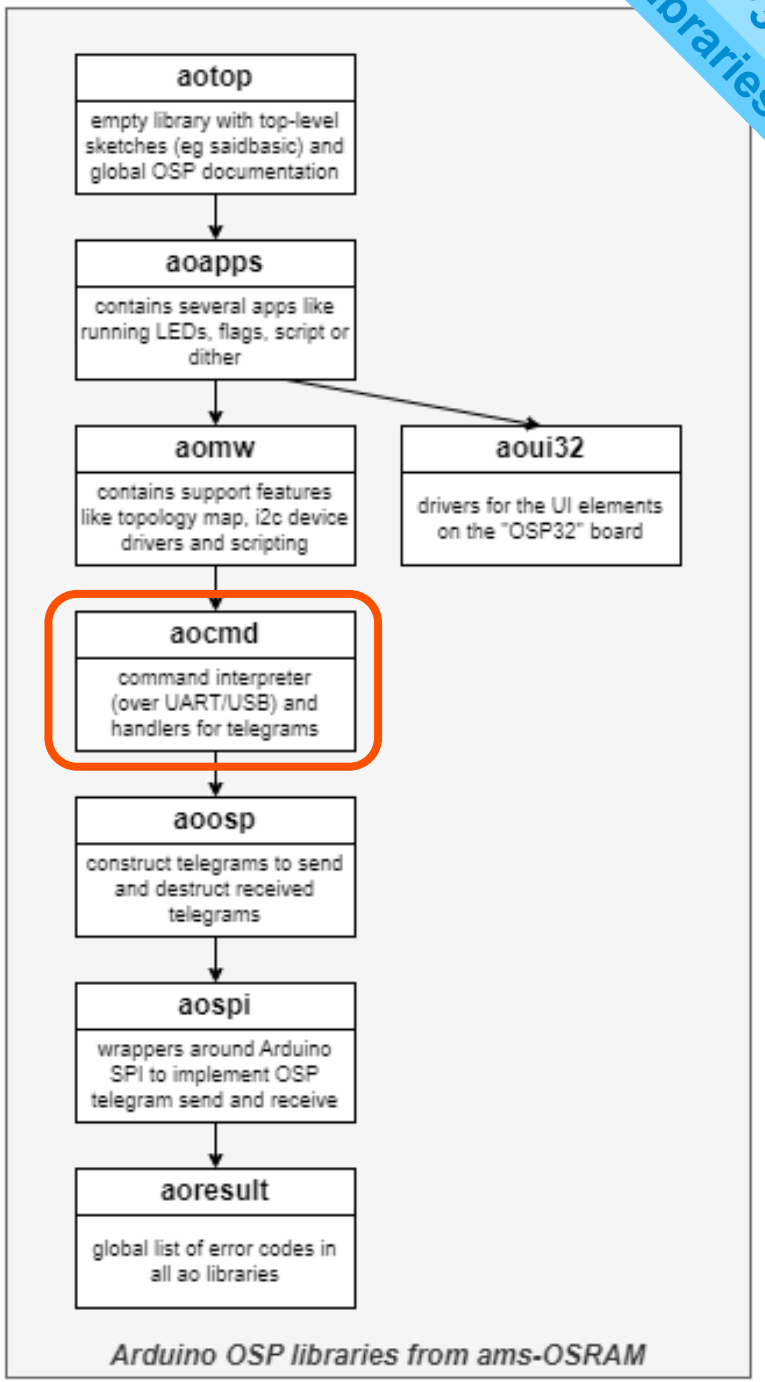
- The OSP32 board has an ESP32S3, with a serial-over-USB
- Several functions in the *aolibs* print over Serial – this is visible in the Arduino IDE

- But aocmd is special, it *receives* characters coming in over serial.
- It implements a command interpreter and several commands (on top of *aoosp*).
- commands: board, echo, file, help, osp, said, topo, version

- This allows connecting a PC interface
  - Connect PC to the OSP32 board (running eg example osplink) to pass commands
  - typically resulting in OSP telegrams being send and received
  - The command interpreter is human centered (like *bash* shall or *cmd.exe*)
  - Useful in the evaluation kit (diagnose, test), not expected in product images

- Bonus 1: ESP32 can store one file (boot.cmd) which is executed at start-up
- Bonus 2: There is a Python proof-of-concept: a PC app that controls an OSP chain

```
// every handler has a registration function
int aocmd_version_register();

// This polls Serial and pushes received chars to the interpreter
void aocmd_cint_pollserial( void );
```



Arduino OSP libraries from ams-OSRAM

Libraries – 5

OSP Middleware aomw

- This library contains an assortment of software features.

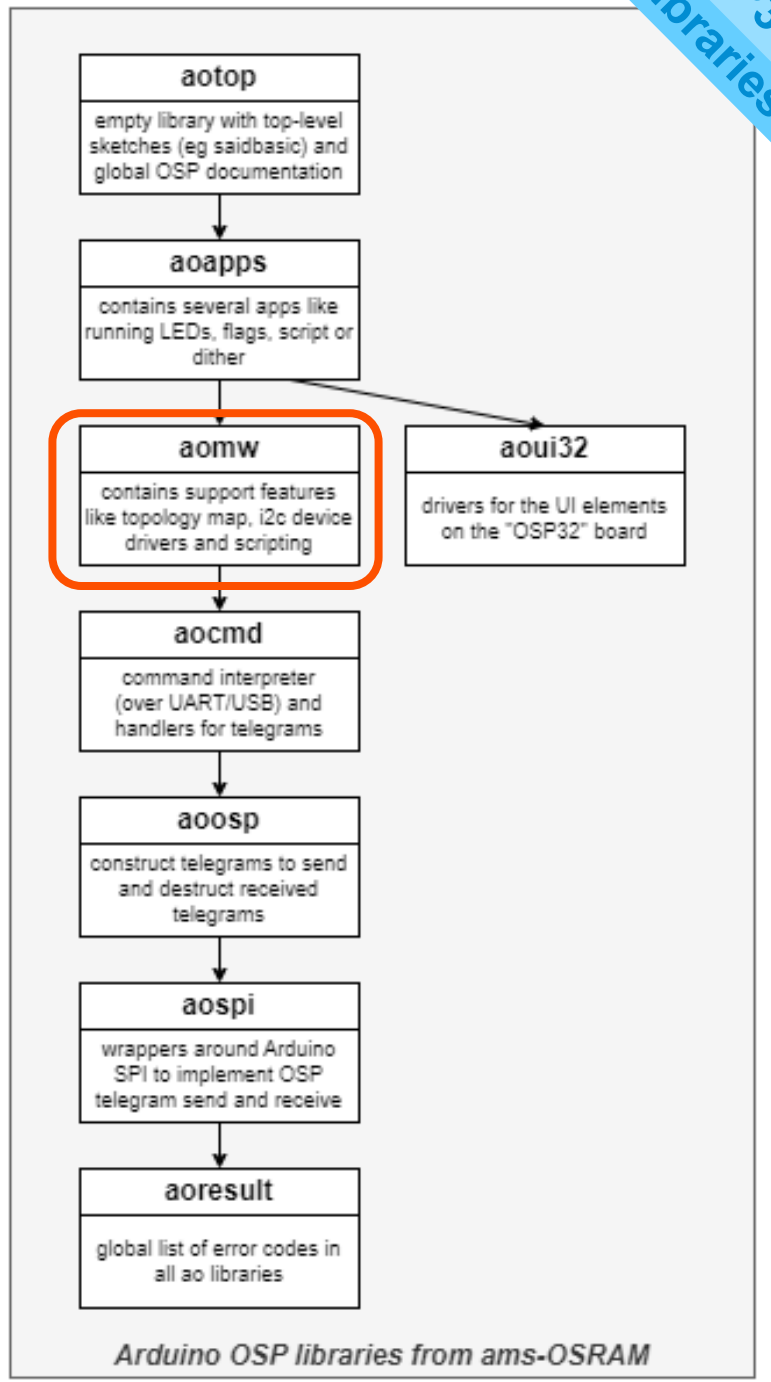
- A topology manager
  - It builds a map of the OSP chain (how many nodes, which is RGBI, which is SAID)
  - making an abstraction of RGB triplets (an RGBI, or on a channel of a SAID)
- A color library to compute the duty cycles of RGB to achieve a target color

- Driver for an I2C EEPROM (on OSP32, SAIDbasic, stick)
- Driver for an I2C I/O-expander

- Paint (country) flags on an OSP chain
- Interpreter for scripted animations

- Useful in making flexible demos, but are not expected in production firmware

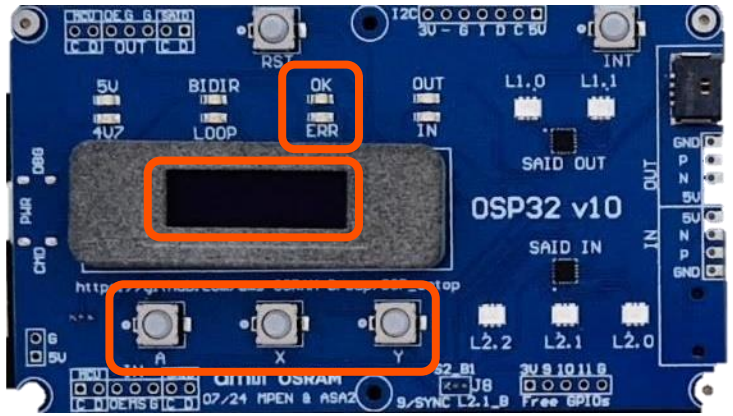
```
aoresult_t aomw_topo_build(); // topo
aoresult_t aomw_topo_settriplet( uint16_t tix, const aomw_topo_rgb_t*rgb ); // topo
void aomw_color_computemix( aomw_color_xyz3_t*source, aomw_color_xyz1_t*target, aomw_color_mix_t*mix ); // color
aoresult_t aomw_eeprom_write(uint16_t addr, uint8_t daddr7, uint8_t raddr, uint8_t *buf, int count ); // eeprom
aoresult_t aomw_iox_led_set( uint8_t leds ); // iox
aoresult_t aomw_flag_painter_dutch(); // flag
aoresult_t aomw_tscript_playframe(); // tscript
```



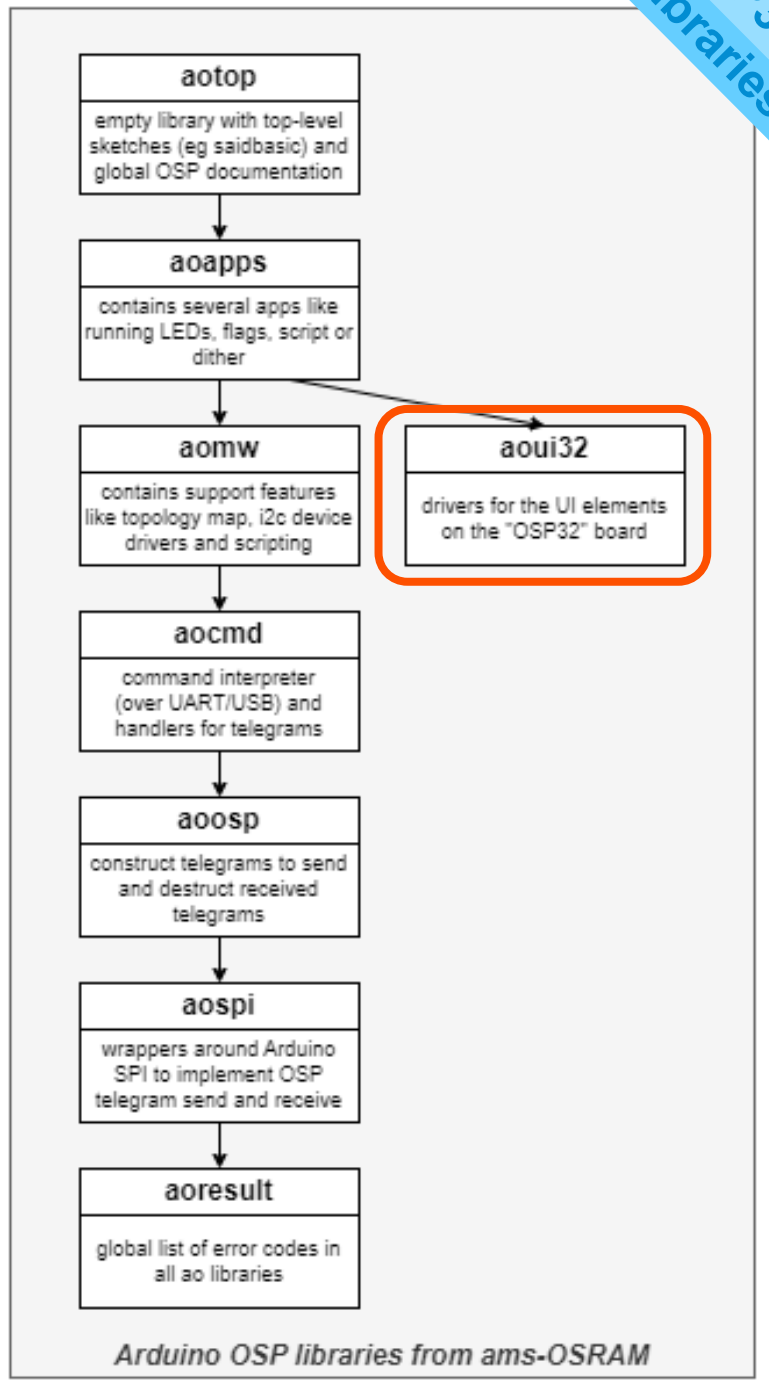
Libraries – 6

OSP UIDriversOSP32 aoui32

- This library contains drivers for the UI elements on the OSP32 board
  - A, X and Y button
  - red (error) and green (ok/heartbeat) signaling LEDs
  - the OLED screen.
- It does not depend on any of the other libraries
- Useful for demos on OSP32, but not expected in production firmware




```
// button
int  aoui32_but_wentdown(int buts);
// led
void aoui32_led_on(int leds);
// OLED
void aoui32_oled_msg(const char * msg);
```



Libraries – 7

OSP ReusableApps aoapps

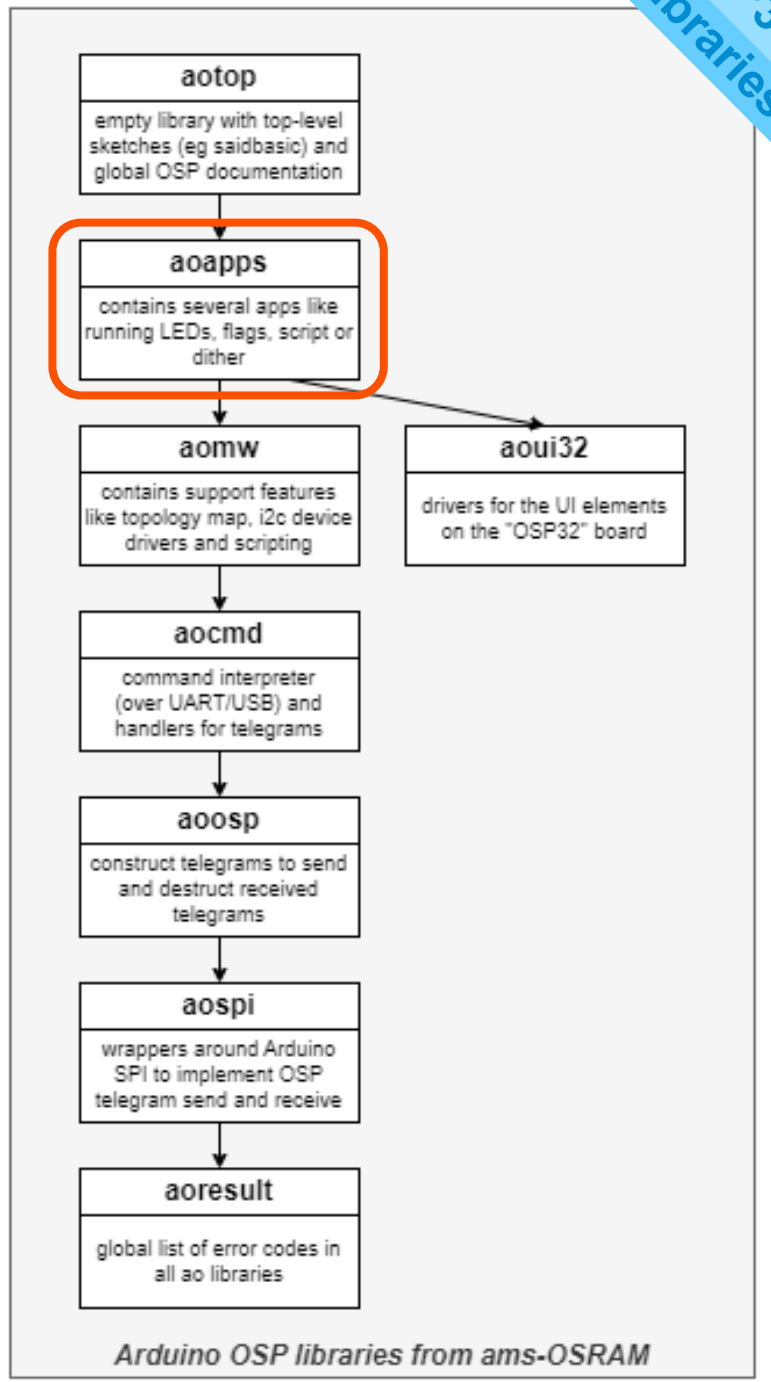
- What is an “app” in context of aolibs?
  - An app is reusable software
  - It performs some animation (running led, animated script)
  - Typically (but not necessarily) runs on top of topo (flexible in connected boards)
  - May use X and Y button

  - One top-level sketch (ie ESP firmware) may include multiple apps
  - One app may be included in multiple top-level sketches

- This library contains apps: running leds, switchable flag, animated script, dithering
- It also contains an app manager (starts and stops apps, eg when A is pressed)

- This library contains demo apps, but those are not expected in production firmware

```
// app manager
void aoapps_mngr_register(char * name, … start, step, stop, …);
// the running leds app
void aoapps_runled_register();
```



Arduino OSP libraries from ams-OSRAM

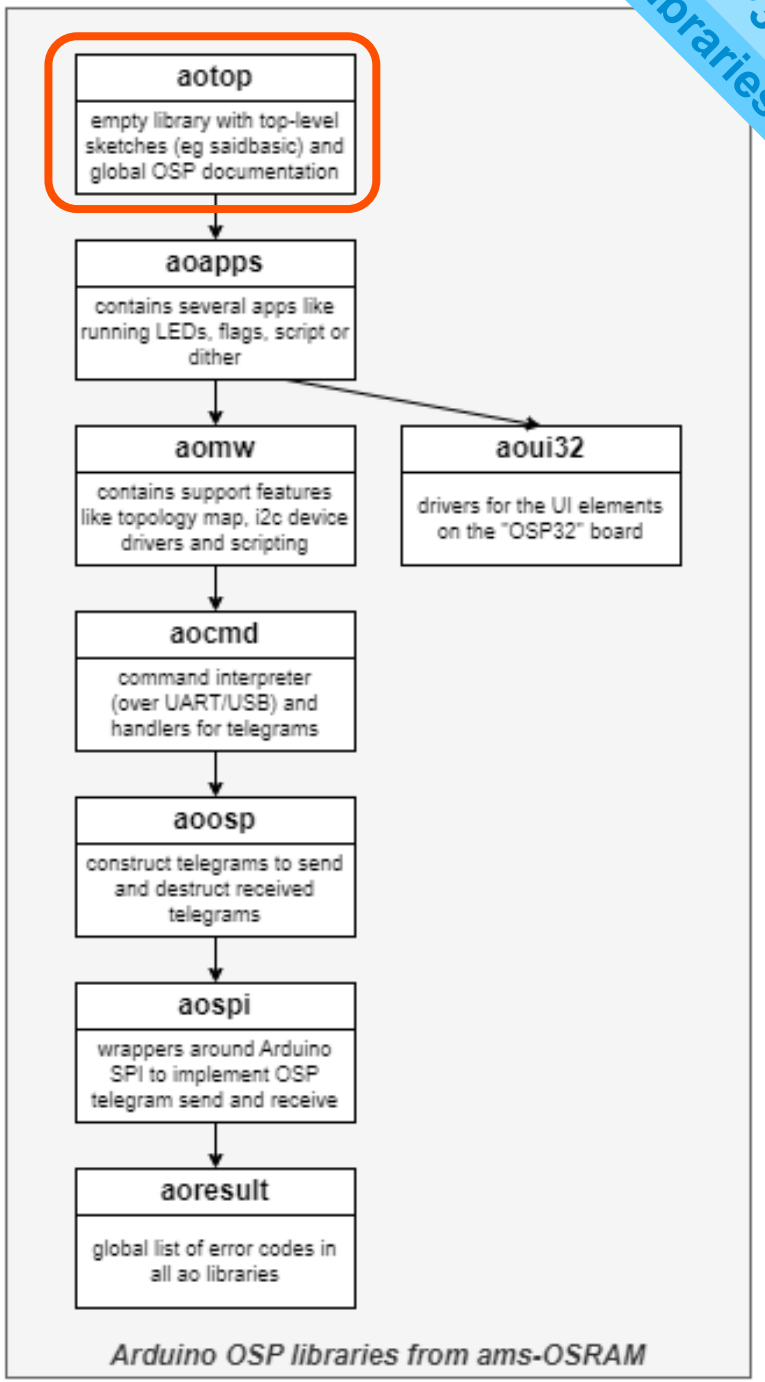
Libraries – 8

OSP ToplevelSketches aotop

aotop
- a repo on GitHub
- technically a library for Arduino (can be installed, has examples)
- NOT a library in reuse sense; it does not have library code

aotop on Github
- is the landing page on GitHub – hyperlinking to all other repos and websites
- contains top-level documentation (like this training; getting started)

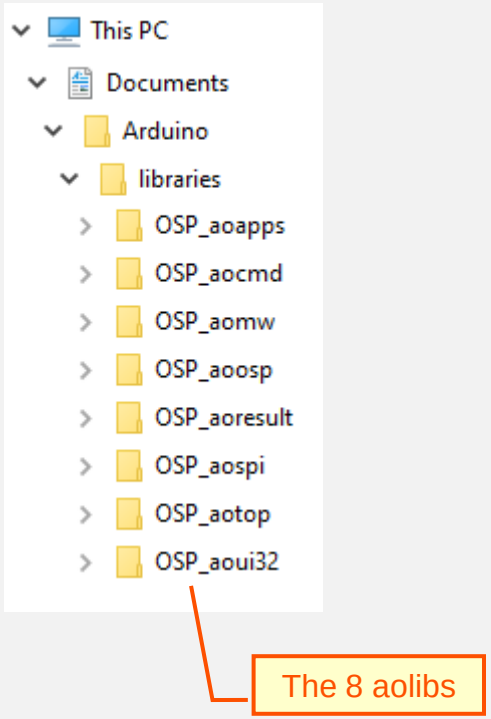
aotop in Arduino
- does contain examples
- examples are code, however not “Sketch> Library” but “File>Examples”
- this code is for top-level sketches for official applications
  - saidbasic, osplink, eepromflasher, and some for this training
- Its “library.properties” file declares it to be dependent on all others
  installing this one via Arduino Library manager installs all others automatically



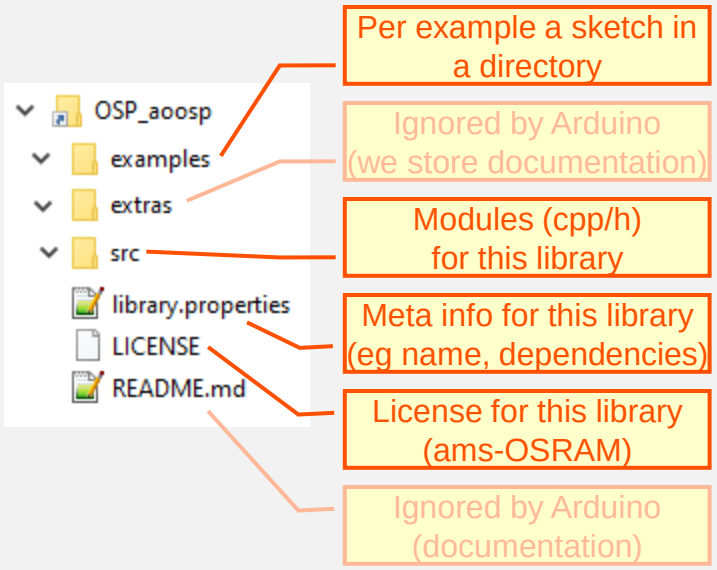
Library directory structure

Libraries, directories, files

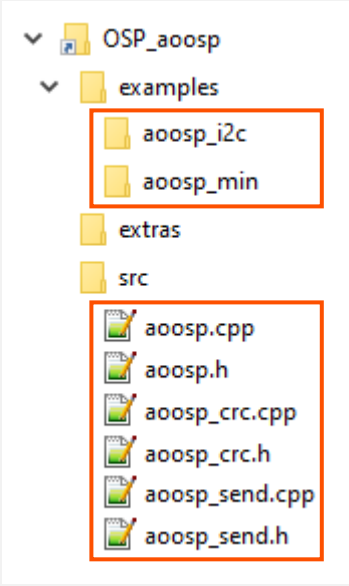
- Downloaded libraries (via Arduino)
- … end up in your filesystem
- One directory per library



- All libraries have the same structure
- Convenient …
- … and dictated by Arduino



- Every library has *examples*
- Every library has *modules*
- Both are prefixed with the (short) library name (our naming convention)



Naming conventions

Libraries, modules, functions

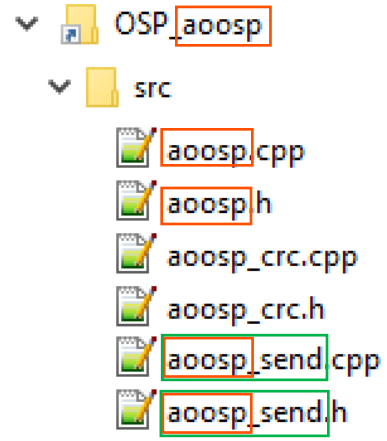
Arduino library name	Prefix
OSP ToplevelSketches aotop	aotop
OSP ReusableApps aoapps	aoapps
OSP UIDriversOSP32 aoui32	aoui32
OSP Middleware aomw	aomw
OSP CommandInterpreter aocmd	aocmd
OSP Telegrams aoosp	aoosp
OSP 2wireSPI aospi	aospi
OSP ResultCodes aoresult	aoresult

Base name

- Arduino uses long names (in its menus) **`OSP ToplevelSketches aotop`**
- GitHub has repository names **`OSP_aotop`**
- The libraries use the short name **`aotop`** as prefix

Rules

- Assume a library has short name is aolll
- Then there is typically a top-level module aolll (aolll.cpp, aolll.h)
- Other a modules of that library have a name like aolll_mmm
- The module files will use aolll_mmm.cpp and aolll_mmm.h
- Public symbols in the module will have a name aolll_mmm_sss




```
// Telegram 04 GOSLEEP - switches the state of the addressed node to sleep.
aoresult_t aoosp_send_gosleep(uint16_t addr );

// Telegram 05 GOACTIVE - switches the state of the addressed node to active.
aoresult_t aoosp_send_goactive(uint16_t addr );
```

Documentation – 1

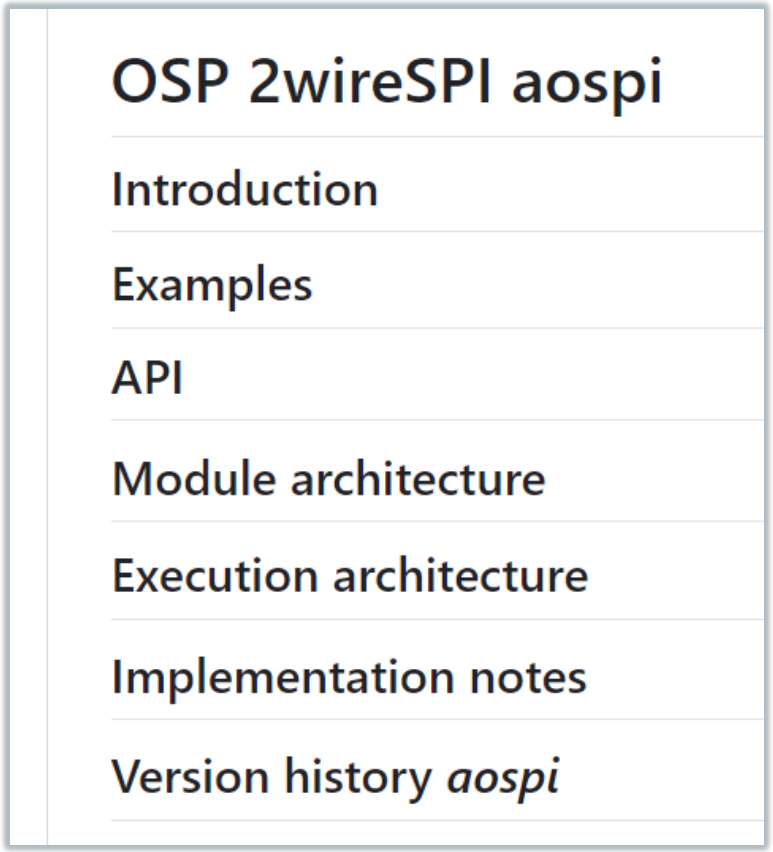
Library wide documentation

Generic documentation

- Top level landing page with links — aotop/readme.md
- Getting started document — aotop/gettingstarted.md
- Other manuals (eg this training) — aotop/extra/manuals
- Introduction to command interpreter — aocmd/readme.md#example-commands

Library specific documentation

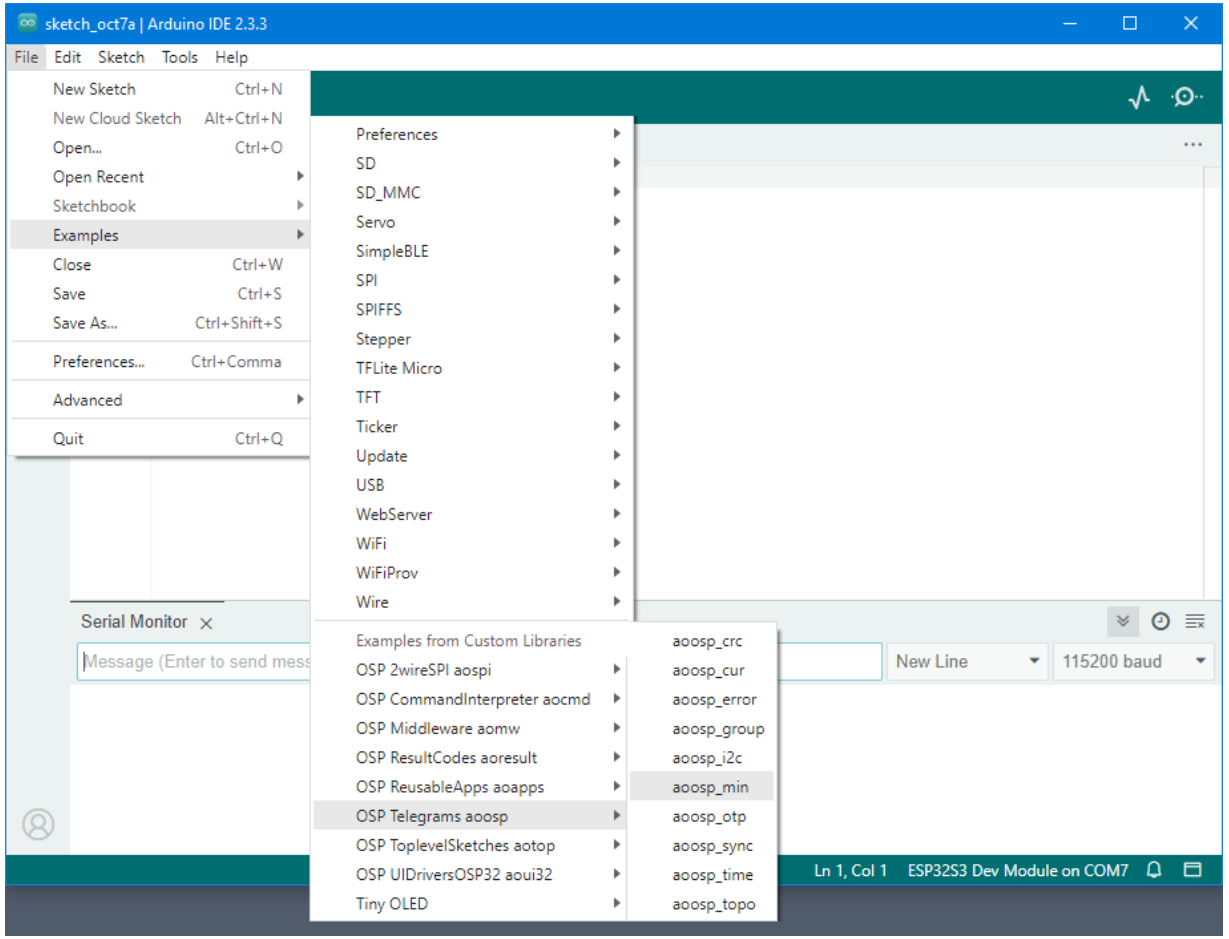
- Every library comes with a readme.md, describing
  - The library as a whole
  - All examples of the library
  - Overview of the API
  - Modules and their interdependencies
  - Sometimes other topics like execution architecture
  - Version history



Documentation – 2

Library examples

- Every library comes with examples
- This is an Arduino feature that we also use
- In the menu File>Examples they are listed, per library
- In total there are more than 30 examples



Documentation – 3

Library API

- All cpp files have “javadoc” documentation for their public functions

```
/*!
    @brief  Sends an GOACTIVE telegram.
            This switches the state of the addressed node to active
            (allowing to switch on LEDs).
    @param  addr
            The address to send the telegram to (unicast),
            (use 0 for broadcast, or 3F0..3FE for group).
    @return aoresult_ok if all ok, otherwise an error code.
    @note   When logging enabled with aoosp_loglevel_set(), logs to Serial.
*/
aoresult_t aoosp_send_goactive(uint16_t addr) {
```

- Note that Arduino 2.x IDE is much smarter than 1.x
  - Right-click > Go to Definition or Ctrl+F12 or Ctrl+LeftMouse all jump from call to definition
  - Unfortunately, it jumps to the header (h) file, not the source (cpp) file
  - The API doc is in the CPP file ☹

If somebody knows how to do documentation better or knows how to easily jump to the implementation, let me know

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

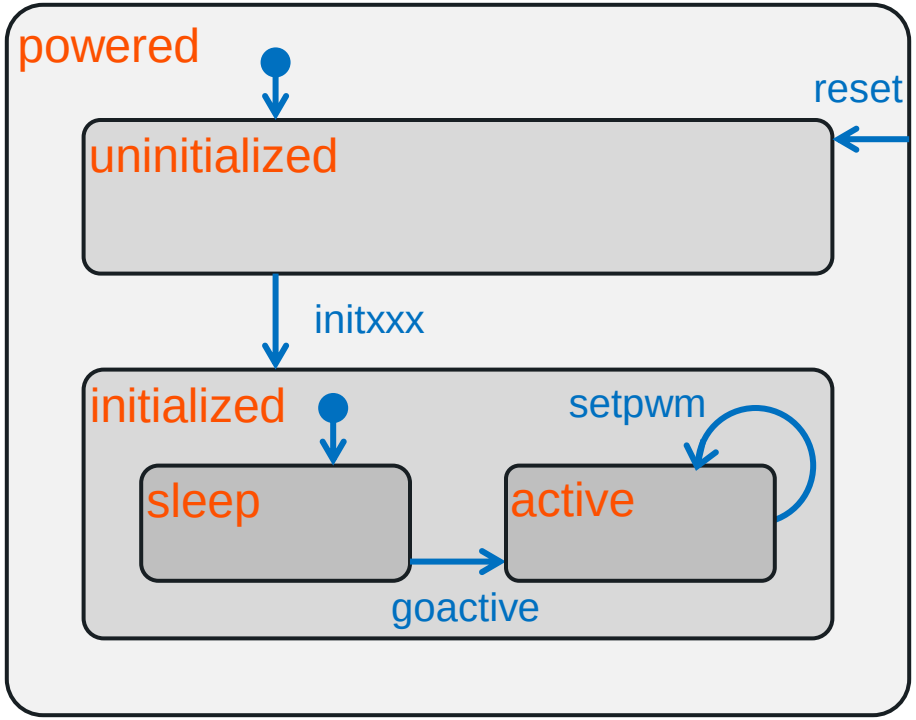
Power states of nodes

Mandated by OSP

- When nodes are powered (PoR / power-on-reset) they are by default in state uninitialized
- In uninitialized a node has no address and can thus not be sent a telegram (specific to that node)

- The initloop or initbidir telegram assigns all nodes an address, they reach state initialized
- When initialized, by default the node is in sleep
- In sleep a node a node can not switch on LEDs

- Send a telegram goactive to move the node to state active
- Then send setpwm to switch on LEDs at some brightness
- At any moment, a reset telegram can be sent, the nodes go back to state uninitialized



This is a simplified state diagram (there are more states and more transitions)
This diagram uses the terms from OSP spec (which differ from SAID – and those are less clear)

Addressing a node – 1

Four flavors in OSP

How can you send a telegram to a node to give it an address, when the node has no address to send the telegram to?

address	cast
000	broadcast
001 – 3EF	1007 unicast addresses
3F0 – 3FE	15 groupcast addresses (group 0 to E)
3FF	reserved

General

- Every telegram has an address, a 10-bit number
- In software usually denoted with 3 hex digits

Unicast

- When a telegram contains a specific node address (001–3EF) this is referred to as unicast
- Only the addressed node reacts to the telegram

0x3EF = 1007

Broadcast

- When a telegram contains the address 000 this is called a broadcast
- All nodes react to that (can thus only be used for commands that do not cause a reply; so GOACTIVE, not READSTAT)

Groupcast

- Every SAID has a 15-bit register (MULT) which determines to which of 15 groups (0..E) that SAID belongs
- With a SETMULT telegram (unicast) this register is written, assigning that SAID to one or more groups
- Addresses 3F0–3FE are the associated 15 group addresses, used for groupcast aka multi cast
- Every SAID that has a matching MULT, reacts (again, no responses allowed)

Serial cast

- See next page

Addressing a node – 2

Flavor 4: serial cast

Serial cast

- In the current OSP specification 6 telegrams are tagged with “serial cast”
- Note: serial cast is a telegram *type* property, not an address property (like other 3 casts)
- A telegram tagged with serial cast *must* be sent to a specific node (001–3EF)
- The telegram might have a payload

- Such a telegram is picked up by the addressed node
- It acts upon the telegram – using the optional payload
- Then it sends out the same telegram type, with an updated payload, to the next node
- or it aborts forwarding and sends a response telegram instead

- Prime example is INITBIDIR with an address
- A node in state uninitialized picks up that telegram
- Assigns itself the passed address
- Increments the address
- Sends out an INITBIDIR with address+1, unless it is the last, then sends a STAT response

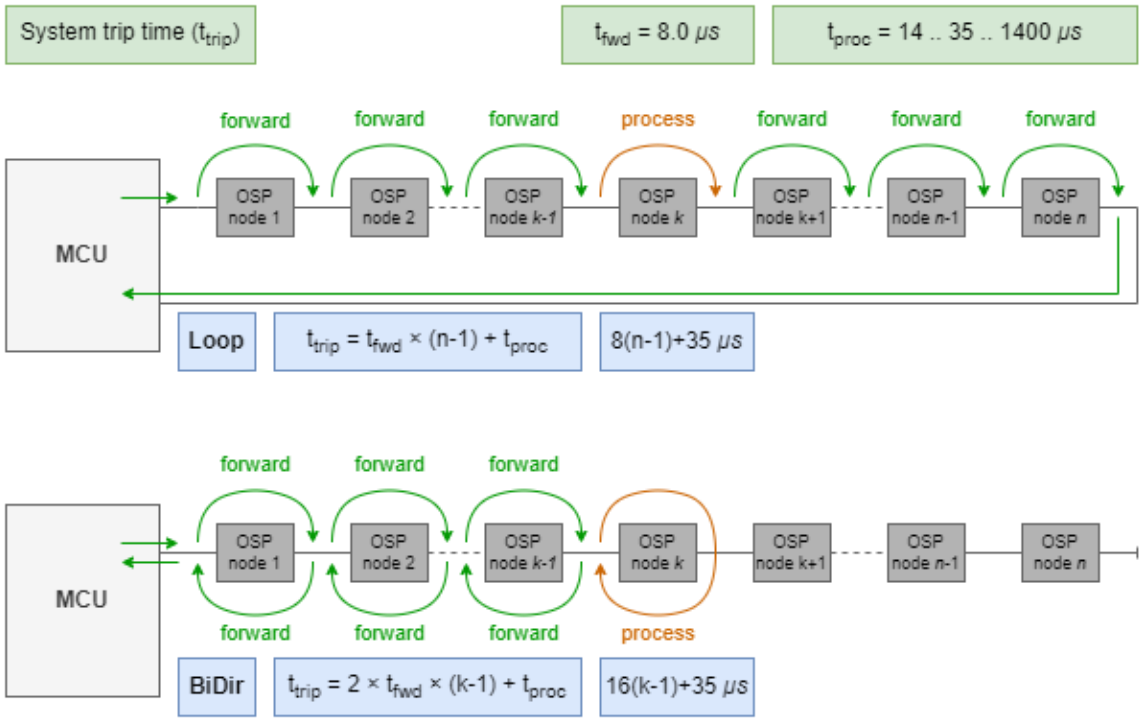
- ASKTINFO computes the maximum temperature of the chain this way

Name	Function	Code	Broadcast / multicast
RESET	Reset	0x00	Yes
CLRERROR	Clear error flag	0x01 0x21	Yes
INITBIDIR	Init Address bidir	0x02	Serial
INITLOOP	Init address loopback	0x03	Serial
GOSLEEP	Enter sleep	0x04 0x24	Yes
GOACTIVE	Go Active	0x05 0x25	Yes
GODEEPSLEEP	Deep Sleep	0x06 0x26	Yes
IDENTIFY	Ask device ID	0x07	No
P4ERRBIDIR	Ping4Err in bidir mode	0x08	Serial
P4ERRLOOP	Ping4Err in loopback	0x09	Serial
ASKTINFO	Max Temp feedback	0x0A	Serial
ASKVINFO	Max Volt feedback	0x0B	Serial
READMULT	Read multicast reg	0x0C	No
SETMULT	Set multicast reg	0X0D 0x2D	No

Telegram timing

Intermezzo

- A telegram is a series of bytes (4 to 12)
- The first bytes contain the address and telegram type
- As soon as a telegram comes in, the node compares its own address with the address in the telegram
- If there is a match (same, broadcast, matching group) the node acts upon the telegram (and maybe forwards)
- If there is no match the telegram starts forwarding to the next node to lose as little time as possible
- The forwarding delay is about 8µs
- Warning: do not send two telegrams with less then 8us delay
- If telegram execution time is extensive (reset, i2cwrite) delay must include execution time
- For reset that is 150µs



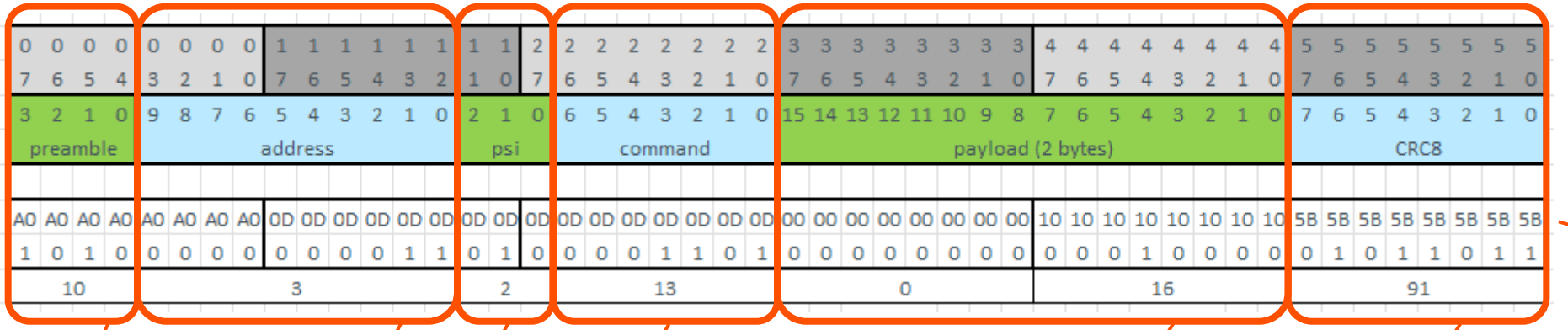
Telegrams – 1

Dissecting a telegram into bytes

Tip: aospi\python\telegram makes similar diagrams or use command “osp fields”

This is an example of a telegram

– SETMULT (0D), to node 003 (hex) with payload (15-bit vector with bit 4 set) 0010 (hex)



Preamble
(always 1010)

Target address
(here 003)

Payload size indication
(here 2 bytes)

Telegram ID
(OD for SETMULT)

Payload
(here 2 bytes 0x0010)

CRC
(OSP spec)

Telegram to send has 6 bytes: A0 0D 0D 00 10 5B

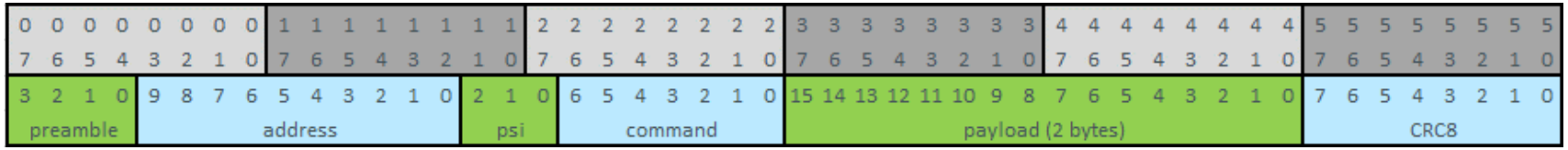
ams OSRAM

Telegrams – 2

Dissecting a telegram into bytes

PSI	payload size (bytes)	comment
0 0 0	0	
0 0 1	1	
0 1 0	2	
0 1 1	3	
1 0 0	4	
1 0 1	5	forbidden
1 1 0	6	
1 1 1	8	exception

- Preamble is mandatory (LVDS uses Manchester, this syncs the clocks)
- Addresses are 10 bits (see previous slides for meaning)
- PSI (payload size indicator) is 3 bits – two exceptions, see table
- Telegram IDs partially standardized by OSP, partially can be chosen by manufacturer
  - Give command “osp info” to a demo to see the list of all currently known commands
  - Note some manufacturers pick same TID with a different meaning (see 4F/SETPWM and 4F/SETPWMCHN)
- Payload contains the arguments for the TID
- CRC following OSP standard (polynomial 0x2F, or $x^8+x^5+x^3+x^2+x^1+x^0$)



Demos with command interpreter allow direct control

- The SETMULT(003,0010) from previous slide can be entered as
- High level command “`osp send 003 setmult 00 10`”
- Low level command “`osp tx A0 0D 0D 00 10 crc`” or for die-hards “`osp tx A0 0D 0D 00 10 5B`”

Telegram types

Commands, read/set registers, responses

The OSP documentation distinguishes commands (one telegram) and registers (two telegrams)

- Command telegrams (e.g. goactive)
  - They are transmitted
  - No response (“answer”) comes back
  - They have a flag (bit 5 of TID) to request an acknowledgement: “goactive with SR (status request)”

- Register access telegrams (e.g. setpwm)
  - There is a register xxx in the node
  - There is a setxxx telegram and a readxxx telegram (they have bit 6 of TID set to *suggest* register)
  - The latter sends a response back

There is no real technical aspect to this convention

We can ignore it

It is not even applied 100% (setmult/readmult looks like a register but does not have bit 6 set)

Response telegrams

Reusing the telegram ID

Response from a node

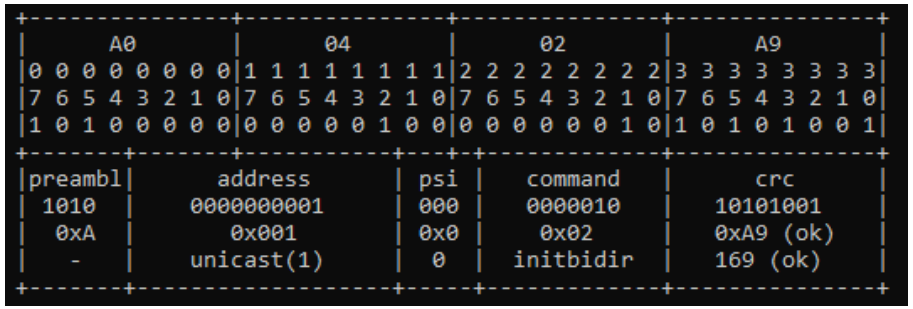
- The address field is no longer the *destination* but rather the *source*
- This is not a problem because no other node is supposed to have the same address, so the response will not be “eaten” along its way to the MCU

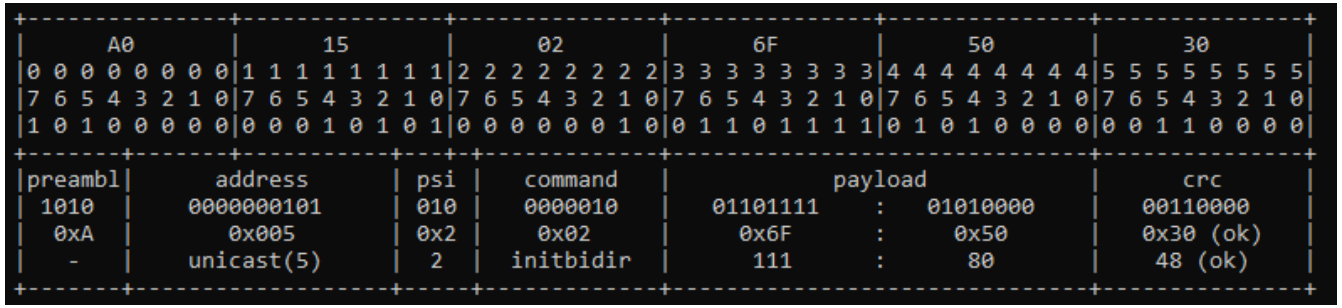
- The TID is copied from the command telegram
- But the payload (and PSI) are response specific

Example on the right

- The command is initbidir(001)
- The response is
  - From node 005
  - Responding to initbidir (TID=2)
  - With two-byte payload (PSI=2)
  - Temperature (6F) and status (50)

Tip: aospi\python\telegram makes these “drawings” or use command “osp fields”





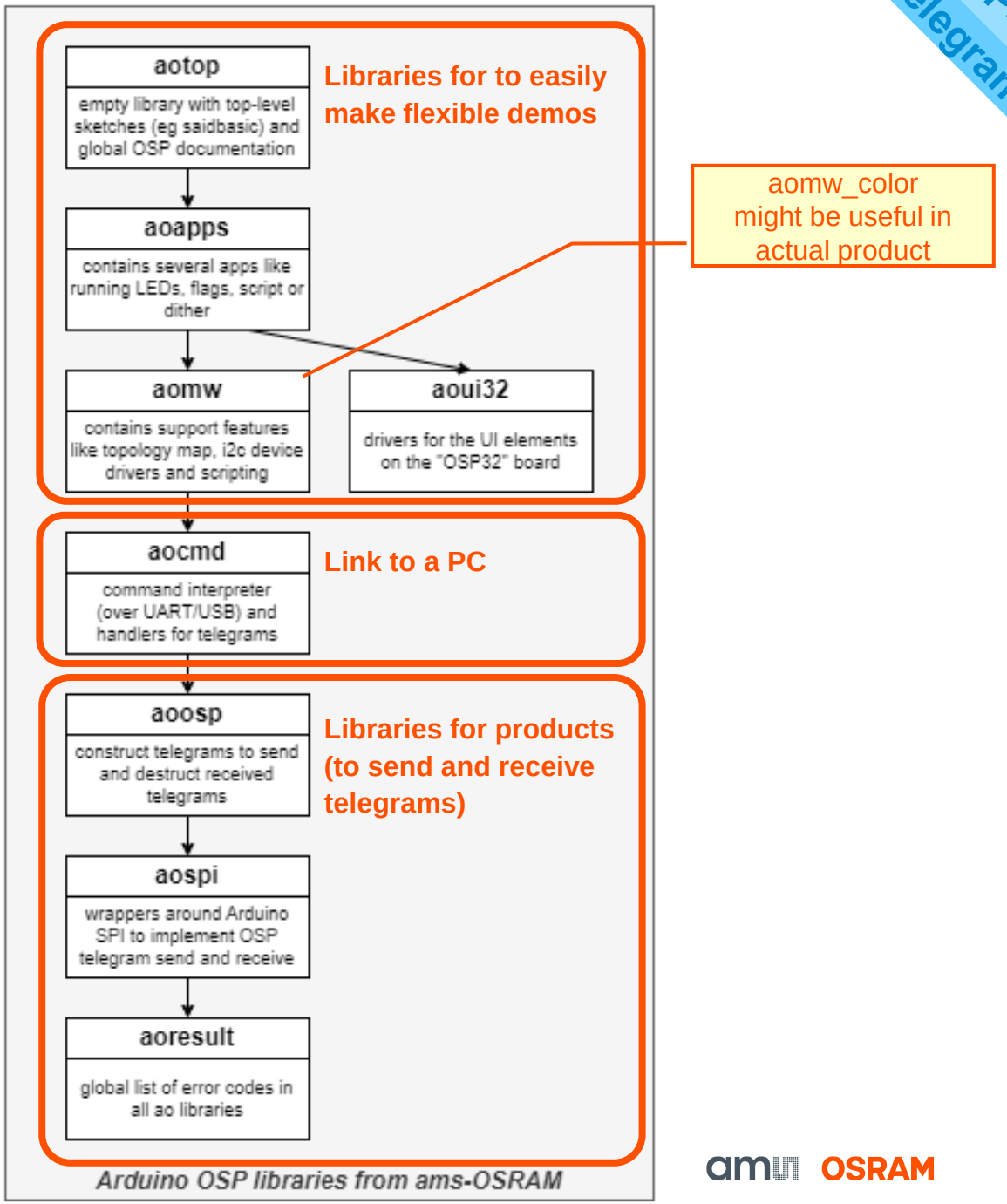
Libraries in detail

Overview

- The upper 4 libraries allow to make (flexible) demos

- The *aocmd* is typically for an eval kit
- It implements OSP32-PC communication
- Via the command interpreter

- The lower 3 libraries implement telegram sending and receiving
- Those are typically part of every OSP product
- Next slides present their details



Libraries in detail – 1

OSP ResultCodes aoresult

A rather empty library

A global list of errors (grouped by typical source)

```
// For detailed meaning, see aoresult_to_str()
typedef enum aoresult_e {
  // General errors
  aoresult_ok               , //  0
  aoresult_assert           , //  1
  aoresult_outargnull       , //  2
  aoresult_outofmem         , //  3
  aoresult_comparefail      , //  4
  aoresult_other            , //  5

  // Errors from aospi
  aoresult_spi_buf          , //  6
  aoresult_spi_noclock      , //  7
  aoresult_spi_length       , //  8

  // Errors for aoosp
  aoresult_osp_arg          , //  9
  aoresult_osp_addr         , // 10
  aoresult_osp_preamble     , // 11
  aoresult_osp_tid          , // 12
  aoresult_osp_size         , // 13
  aoresult_osp_psi          , // 14
  aoresult_osp_crc          , // 15
  aoresult_osp_nosr         , // 16

  // Errors on OSP system
  aoresult_sys_cabling      , // 17
  aoresult_sys_id           , // 18
  aoresult_sys_wrongtopo    , // 19
  aoresult_sys_nodecfg      , // 20

  // Errors for attached devices
  aoresult_dev_noi2cbridge , // 21
  aoresult_dev_noi2cdev     , // 22
  aoresult_dev_i2ctimeout   , // 23
  aoresult_dev_i2cnack      , // 24

  aoresult_numresultcodes    // 25 keep this as last
} aoresult_t;


// Converts an aoresult_t error code to a string.
const char * aoresult_to_str(aoresult_t result, int verbose=0);


#define AORESULT_ASSERT(cond) \
  if( !(cond) ) { \
    Serial.printf("ASSERT( %s ) in %s line %d\n", #cond, __FILE__, __LINE__ ); \
    /* asm("break.n 1"); */ while(1) /*wait*/ ; \
  }
```

```
const char * aoresult_to_str(aoresult_t result, int verbose) {
  switch( result ) {
    case aoresult_ok               : return verbose==0 ? "ok"               : "Success (no error)";
    case aoresult_outargnull       : return verbose==0 ? "outargnull"       : "An output argument is NULL";
    ...
}
```

Helper function mapping a result code to a short (default) or long string (verbose=1)

To help find errors, many API functions use asserts. Here is the implementation.

Libraries in detail – 2

OSP 2wireSPI aospi

This library needs an init
(to setup all pins and the SPI blocks)

```
// Initializes the SPI OUT and IN controllers and their support pins.
void aospi_init();


// Sends the `txsize` bytes in buffer `tx` to the first OSP node.
aoresult_t aospi_tx(const uint8_t * tx, int txsize);
// Sends the `txsize` bytes in buffer `tx` to the first OSP node.
// Waits for a response telegram and stores those bytes in buffer `rx` with size `rxsize`.
aoresult_t aospi_txrx(const uint8_t * tx, int txsize, uint8_t * rx, int rxsize, int *actsize=0);


// Sets the direction mux so that the last OSP node is connected to the SPI slave (for an OSP chain using Loop).
void aospi_dirmux_set_loop();
// Sets the direction mux so that the first OSP node is connected to the SPI slave (for an OSP chain using BiDir).
void aospi_dirmux_set_bidir();
```

If you compose your own telegram, you can pass that byte array for transmission

If you compose your own telegram, and expect a response use this instead

To be able to receive a response
(before calling aospi_txrx),
the mux must be set correctly
(remember, there is a signaling LED)

There are other functions:

- To *observe* mux state
- To measure round trip time
- Stats for number of tx'x and rx's
- To test the signaling LEDs

Libraries in detail – 3

OSP Telegrams aoosp

```
// Initializes the aoosp library.
void aoosp_init();

// Telegram 00 RESET - resets all nodes in the chain (all "off"; they also lose their address).
aoresult_t aoosp_send_reset(uint16_t addr );
// Telegram 02 INITBIDIR - assigns an address to each node; also configures all nodes for BiDir.
aoresult_t aoosp_send_initbidir(uint16_t addr, uint16_t * last, uint8_t * temp, uint8_t * stat );
// Telegram 03 INITLOOP - assigns an address to each node; also configures all nodes for Loop.
aoresult_t aoosp_send_initloop(uint16_t addr, uint16_t * last, uint8_t * temp, uint8_t * stat );
// Telegram 05 GOACTIVE - switches the state of the addressed node to active.
aoresult_t aoosp_send_goactive(uint16_t addr );

// Telegram 01 CLRERROR - clears the error flags of the addressed node.
aoresult_t aoosp_send_clrerror(uint16_t addr);
// Telegram 07 IDENTIFY - asks the addressed node to respond with its ID.
aoresult_t aoosp_send_identify(uint16_t addr, uint32_t * id );
// Telegram 0D SETMULT - assigns the addressed node to zero or more of the 15 groups.
aoresult_t aoosp_send_setmult(uint16_t addr, uint16_t groups);
// Telegram 40 READSTAT - asks the addressed node to respond with its (system) status.
aoresult_t aoosp_send_readstat(uint16_t addr, uint8_t * stat);
// Telegram 51 SETCURCHN - configures the current levels of the addressed node for the specified channel
aoresult_t aoosp_send_setcurchn(uint16_t addr, uint8_t chn, uint8_t flags, uint8_t rcur, uint8_t gcur, uint8_t bcur);


// Telegram 4F (variant 0) SETPWM - configures the PWM settings of the addressed node (single channel nodes).
aoresult_t aoosp_send_setpwm(uint16_t addr, uint16_t red, uint16_t green, uint16_t blue, uint8_t daytimes );
// Telegram 4F (variant 1) SETPWMCHN - configures the PWM settings of one channel of the addressed node.
aoresult_t aoosp_send_setpwmchn(uint16_t addr, uint8_t chn, uint16_t red, uint16_t green, uint16_t blue );
```

Always start by broadcasting a reset

Next, assign all nodes an address (bidir/loop)

Last node responds with its address (and temp/stat)

Next, switch a node to active (otherwise no light)

Example telegrams (many omitted)

Set the PWM values of the R/G/B drivers
3-bit daytimes sets drive current(10/50mA)

Note that a SAID has three channels
(three RGBs, so setpwm**chn** is needed)

For SAID the *driver current* is
set via a separate telegram

Warning, SAID comes out of reset with “over-voltage” error.
This flag must be cleared, or SAID refuses to goactive

There are other functions:

- Compute CRC
- Enable/disable logging
- Pretty print telegram fields
- Exec macros with multiple telegrams

Libraries in detail – 4

Omitted

Scope

- OSP nodes have many features and thus many telegram types
- Too many for this training
- However, there are several example demoing those features
- Find them in aoosp/examples

Examples

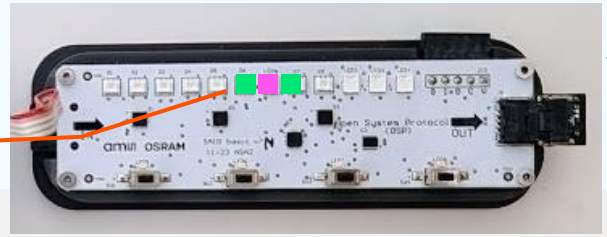
- Bringup
- Minimalistic LED on
- CRC computation
- Setting drive current
- Error detecting and trapping
- Assigning nodes to a group (multicast)
- I2C (this is part of the training)
- Timing experiment
- Using the ADC (on a LED or on an external source)
- Synchronous PWM change (SYNC)
- LED open/short behavior
- Clustering drivers
- Serial cast (ASKTINFO)
- OTP (one-time-programmable memory with SAID configuration)
- Burning a new image in the OTP
- Topology

Assignment – Training1 – green/magenta/green

“Hello world” on OSP

In step 0: magenta RGBI

In step 3: two greens



- From *aotop* build *training1* using “ESP32S3 Dev Module”
- Flash/upload it to the ESP (use USB “CMD”)
- Connect the SAIDbasic board in BiDir mode
- Don’t forget the terminator

0

```
void setup() {
  Serial.begin(115200);
  Serial.printf("\n\ntraining1.ino - green/magenta/green\n");

  aospi_init();
  aoosp_init();

  // aoosp_loglevel_set( aoosp_loglevel_tele );
  anim();
  // Serial.printf("tx %d rx %d\n", aospi_txcount_get(), aospi_rxcount_get() );
}
```

Check the logging

- Uncomment aoosp_loglevel_set()
- and/or aospi_txcount_get()

1

reset has 150us exec time

```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose their address).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_bidir();
  result= aoosp_send_initbidir(0x001, &last, &temp, &stat);
  Serial.printf("initbidir(001) %s last %03X\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );

  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004) %s\n", aoresult_to_str(result) );
}
```

Switch system to Loop mode

- Replace terminator by cable
- Use aoosp_send_initloop()
- Use aospi_dirmux_set_loop()

2

Make led to the left and right of the magenta RGBI turn green

- Use aoosp_send_setpwmchn() – pass correct channel
- Use aoosp_send_goactive() – don’t forget to clear error first

3

```
    001  | 002  003  004  005  006  007  008  |
MCU SAID | SAID SAID RGBI SAID RGBI RGBI RGBI | term
   OSP32 | SAIDbasic                          |
```


ams OSRAM

2 Switch system to Loop mode

“Hello world” on OSP

Don’t forget to set the mux

```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose their address).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_bidir();
  result= aoosp_send_initbidir(0x001, &last, &temp, &stat);
  Serial.printf("initbidir(001) %s last %03X\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );

  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );
}
```

```
training1.ino - green/magenta/green
spi: init
osp: init

reset(000) ok
initbidir(001) ok last 008
goactive(004) ok
setpwm(004,magenta) ok
```

```
   001  | 002  003  004  005  006  007  008  |
MCU SAID | SAID SAID RGBI SAID RGBI RGBI RGBI | term
   OSP32 | SAIDbasic                         |
```

```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose t      dress).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_loop();
  result= aoosp_send_initloop(0x001, &last, &temp, &stat);
  Serial.printf("initloop(001) %s last %03X\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );

  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );
}
```

```
training1.ino - green/magenta/green
spi: init
osp: init

reset(000) ok
initloop(001) ok last 009
goactive(004) ok
setpwm(004,magenta) ok
```

One more node in chain

```
   001  | 002  003  004  005  006  007  008  | 009
MCU SAID | SAID SAID RGBI SAID RGBI RGBI RGBI | SAID
   OSP32 | SAIDbasic                         | OSP32
```

3 Make led to the left and right of the magenta RGBI turn green

“Hello world” on OSP


```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose their address).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_loop();
  result= aoosp_send_initloop(0x001, &last, &temp, &stat);
  Serial.printf("initloop(001) %s last %03X\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );
  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );

  // Clear the error flags of node 003 (unicast)
  result= aoosp_send_clrerror(0x003);
  Serial.printf("clrerror(003) %s\n", aoresult_to_str(result) );
  // Switch the state node 003 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x003);
  Serial.printf("goactive(003) %s\n", aoresult_to_str(result) );
  // Set three PWM values of SAID.CH2 at 003 (unicast) to dim green
  result= aoosp_send_setpwmchn(0x003, 2, 0x0000/*red*/, 0x05FF/*green*/, 0x0000/*blue*/);
  Serial.printf("setpwmchn(003,2,green) %s\n", aoresult_to_str(result) );

  // Clear the error flags of node 005 (unicast)
  result= aoosp_send_clrerror(0x005);
  Serial.printf("clrerror(005) %s\n", aoresult_to_str(result) );
  // Switch the state node 005 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x005);
  Serial.printf("goactive(005) %s\n", aoresult_to_str(result) );
  // Set three PWM values of SAID.CH0 at 005 (unicast) to dim green
  result= aoosp_send_setpwmchn(0x005, 0, 0x0000/*red*/, 0x05FF/*green*/, 0x0000/*blue*/);
  Serial.printf("setpwmchn(005,0,green) %s\n", aoresult_to_str(result) );
}
```

```
   001  | 002  003  004  005  006  007  008  | 009
MCU SAID | SAID SAID RGBI SAID RGBI RGBI RGBI | SAID
  OSP32  | SAIDbasic                          | OSP32
```

```
// Get the state of node 003 to find SLEEP and OV_FLAG.
result= aoosp_send_readstat(0x003,&stat);
Serial.printf("getstatus(003) %s %02X\n", aoresult_to_str(result),stat );

→ getstatus(003) ok 50 → 0b_0101_0000 → SLEEP + OV_FLAG
```

SAIDs have the V flag (over-voltage) after reset, preventing them from going active; clear the error flags of 003 and 005.

SAIDs have three channels. We can not use setpwm(), we must use setpwmchn().

For node 003 we must use the left-most channel.

For node 005 we must use the right-most channel.

3 Shorten the code

“Hello world” on OSP


```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose their address).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_loop();
  result= aoosp_send_initloop(0x001, &last, &temp, &stat);
  Serial.printf("initloop(001) %s last %03X\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );
  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );

  // Clear the error flags of node 003 (unicast)
  result= aoosp_send_clrerror(0x003);
  Serial.printf("clrerror(003) %s\n", aoresult_to_str(result) );
  // Switch the state node 003 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x003);
  Serial.printf("goactive(003) %s\n", aoresult_to_str(result) );
  // Set three PWM values of SAID.CH2 at 003 (unicast) to dim green
  result= aoosp_send_setpwmchn(0x003, 2, 0x0000/*red*/, 0x05FF/*green*/, 0x0000/*blue*/);
  Serial.printf("setpwmchn(003,2,green) %s\n", aoresult_to_str(result) );

  // Clear the error flags of node 005 (unicast)
  result= aoosp_send_clrerror(0x005);
  Serial.printf("clrerror(005) %s\n", aoresult_to_str(result) );
  // Switch the state node 005 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x005);
  Serial.printf("goactive(005) %s\n", aoresult_to_str(result) );
  // Set three PWM values of SAID.CH0 at 005 (unicast) to dim green
  result= aoosp_send_setpwmchn(0x005, 0, 0x0000/*red*/, 0x05FF/*green*/, 0x0000/*blue*/);
  Serial.printf("setpwmchn(005,0,green) %s\n", aoresult_to_str(result) );
}
```

```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose their address).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_loop();
  result= aoosp_send_initloop(0x001, &last, &temp, &stat);
  Serial.printf("initloop(001) %s last %03X\n", aoresult_to_str(result), last );

  // Clear the error flags of all (broadcast)
  result= aoosp_send_clrerror(0x000);
  Serial.printf("clrerror(000) %s\n", aoresult_to_str(result) );

  // Switch the state of all nodes (broadcast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x000);
  Serial.printf("goactive(000) %s\n", aoresult_to_str(result) );

  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );

  // Set three PWM values of SAID.CH2 at 003 (unicast) to dim green
  result= aoosp_send_setpwmchn(0x003, 2, 0x0000/*red*/, 0x05FF/*green*/, 0x0000/*blue*/);
  Serial.printf("setpwmchn(003,2,green) %s\n", aoresult_to_str(result) );

  // Set three PWM values of SAID.CH0 at 005 (unicast) to dim green
  result= aoosp_send_setpwmchn(0x005, 0, 0x0000/*red*/, 0x05FF/*green*/, 0x0000/*blue*/);
  Serial.printf("setpwmchn(005,0,green) %s\n", aoresult_to_str(result) );
}
```

Dot no unicast each node, use broadcast

Libraries in detail – 4

OSP Telegrams aoosp - continued

The aoosp_exec module implements “macros”: multiple telegrams to achieve one user action

The resetinit performs reset then init, but tries both initloop and initbidir (and sets the dirmux)

I2C is for a next chapter

```
// Sends RESET and INIT telegrams, auto detecting BiDir or Loop.
aoresult_t aoosp_exec_resetinit(uint16_t *last=0, int *loop=0);


// Reads the I2C_BRIDGE_EN bit in OTP (mirror).
aoresult_t aoosp_exec_i2cenable_get(uint16_t addr, int * enable);
// Checks if the SAID has an I2C bridge, if so, powers the I2C bus.
aoresult_t aoosp_exec_i2cpower(uint16_t addr);


// Writes to an I2C device connected to a SAID with I2C bridge..
aoresult_t aoosp_exec_i2cwrite8(uint16_t addr, uint8_t daddr7, uint8_t raddr, const uint8_t *buf, uint8_t count);
// Reads from an I2C device connected to a SAID with I2C bridge.
aoresult_t aoosp_exec_i2cread8(uint16_t addr, uint8_t daddr7, uint8_t raddr, uint8_t *buf, uint8_t count);
```

```
static void anim( ) {
  // Reset all nodes (broadcast) in the chain (all "off"; they also lose their address).
  result= aoosp_send_reset(0x000); delayMicroseconds(150);
  Serial.printf("reset(000) %s\n", aoresult_to_str(result) );

  // Assign an address to each node (starting from 1, serialcast).
  aospi_dirmux_set_loop();
  result= aoosp_send_initloop(0x001, &last, &temp, &stat);
  Serial.printf("initloop(001) %s last %03X\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );

  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );
}
```

```
static void anim( ) {
  // Reset and init all nodes
  result= aoosp_exec_resetinit(&last);
  Serial.printf("resetinit() %s %d\n", aoresult_to_str(result), last );

  // Switch the state node 004 (unicast) to active (allowing to switch on LEDs).
  result= aoosp_send_goactive(0x004);
  Serial.printf("goactive(004) %s\n", aoresult_to_str(result) );

  // Set three PWM values of RGBI at 004 (unicast) to dim magenta (all 3 in nightmode)
  result= aoosp_send_setpwm(0x004, 0x04FF/*red*/, 0x0000/*green*/, 0x08FF/*blue*/, 0b000);
  Serial.printf("setpwm(004,magenta) %s\n", aoresult_to_str(result) );
}
```

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

I2C in SAID – 1

I2C master on channel 2

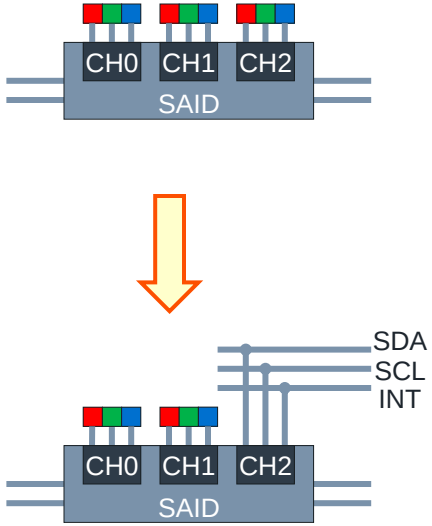
0D	CH_clustering[2:0]			haptic_driver	SPI_mode	sync_pin_en	star_net_en	i2c_bridge_en
0E					otp_addr_en	star_net_otp_addr[2:0]		
0F								

- The SAID comes with 3 channels (CH0, CH1, CH2)
  - Each with 3 drivers (so 9 pads)
  - For 3 times an RGB LED (“3 triplets”)

- The third channel (CH2) can be reconfigured for I2C
  - Resulting in 2 triplets plus I2C
  - Clock (SCL) plus data (SDA) and even an interrupt line (INT)

- Reconfigure is done by setting bit **`i2c_bridge_en`** (0D.0) in the SAID OTP to 1
  - Advise: use bit in the actual OTP
  - Setting the bit in the OTP mirror (RAM) happens to work (for this bit)
  - [note: use the PWM driver to power the I2C bus]

- SAID then acts as an I2C master on that bus
  - 5V (is 3V3 and even 1V8 tolerant)
  - Supports 100kHz (even 78kHz) and 400kHz (even 874kHz)
  - I2C transactions are wrapped in OSP telegrams (so subset of I2C is available)



I2C in SAID – 2

How to use the I2C in SAID

	Bit	Field	Type	Reset	Description
stat	7	I2C_INTERRUP	R	0	This is the I2C interrupt bit
conf	6	I2C_12_BIT_ADDR	RW	0	This enables the 12 bits address mode for I2C
stat	5	NACK_ACK	R	0	This is the nack/ack bit. It will be 1 after that an ack is received.
stat	4	I2C_BUSY	R	0	This is the busy flag
conf	3 … 0	I2C_SPEED[3:0]	RW	0xC	This value selects the divisor factor for I2C speed.

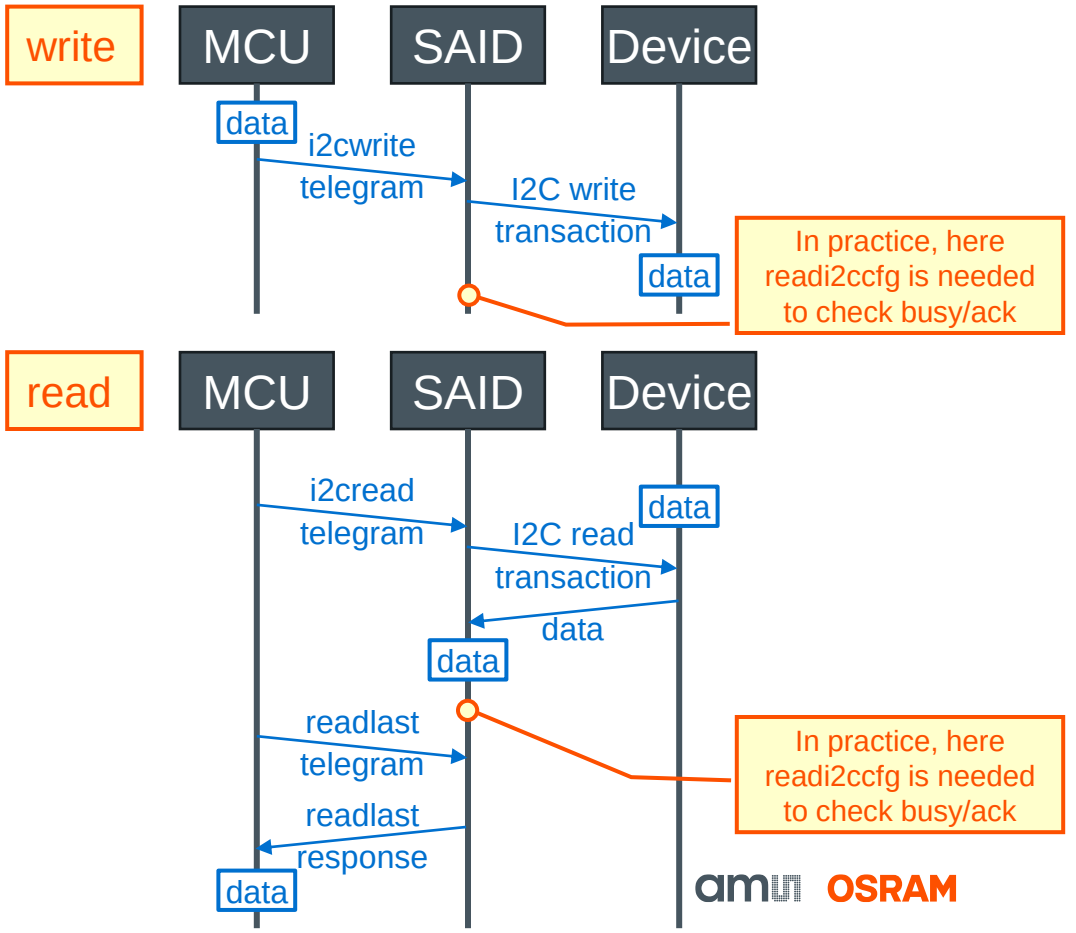
The SAID has one configure/status register

- Under the name I2CCFG (sometimes also named I2CSTAT – same thing)
- To configure the bus speed (and 12 bit-mode – forget it)
- To obtain transaction status (eg acknowledge received?)
- As usual two telegrams for one register: **`readi2ccfg`**, **`seti2ccfg`**

The SAID has two telegrams which invoke an I2C transaction

- **`i2cwrite`** and **`i2cread`**
- The write telegram causes the SAID to master a write transaction on the bus (writing to a connected I2C device)
- The read telegram causes the SAID to master a read transaction on the bus (reading from a connected I2C device)

- Since the latter is slow, there is a third telegram **`readlast`** to get the “cached” read results from SAIDs **`i2cread`** into the root MCU



I2C protocol refresher

Transactions

An I2C **transaction** consists of
- a START condition,
- followed by one or more **segments** separated by (“repeated”) START conditions (zero segments technically probably works, but is useless so not discussed here)
- A STOP condition

Each segment in an I2C transaction transfers **bytes**
- The master always initiates the transfer, from master to device (“write”) or from device to master (“read”)
- Examples
  - A transaction can consist of one segment writing to device A (very typical)

  - A transaction can consist of two segments the first writing to device A the second reading from A (very typical)

  - One transaction could consist of three segments: reading from A, writing to B and writing to A (theoretically allowed)
  - Many more wild combinations of segments are possible
  - SAID only supports the first two transactions (A:write and A:write+A:read) in practice this is not a restriction

A read segment is nearly always preceded by a write segment; it indicates **what** to read

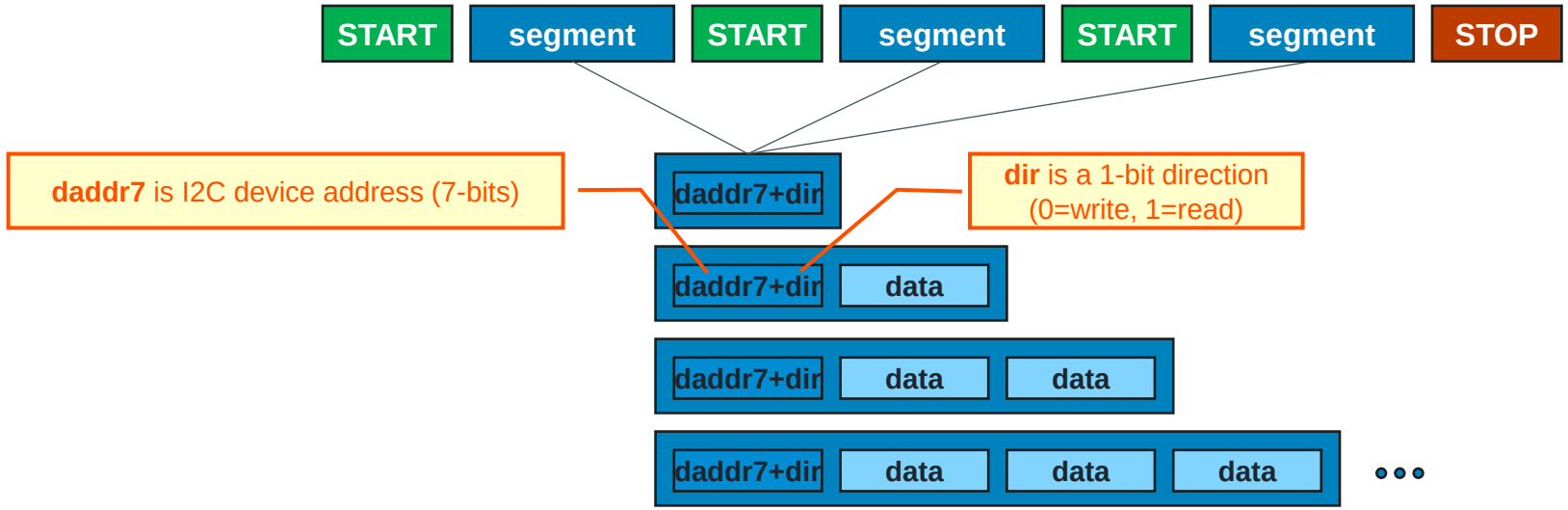
The notion of transactions in I2C is for multi-master busses: a transaction is **atomic**, a second master can not interrupt

I2C protocol refresher

Segments (read or write)

A segment consists of (transfers of) one or more **bytes**

- The first byte is mandatory, it consists of a 7 bits device address, followed by a 1-bit direction (read/$\overline{\text{write}}$)
- All other bytes (**zero** or more) are optional data bytes (to or from device).
- Every byte is transferred in 9 clock ticks, clock tick 9 is the ack/nack bit
- I2C just transfers (whole) bytes; there is no notion of other “word sizes” (12 bits, 16 bits); there is **no notion what bytes mean**



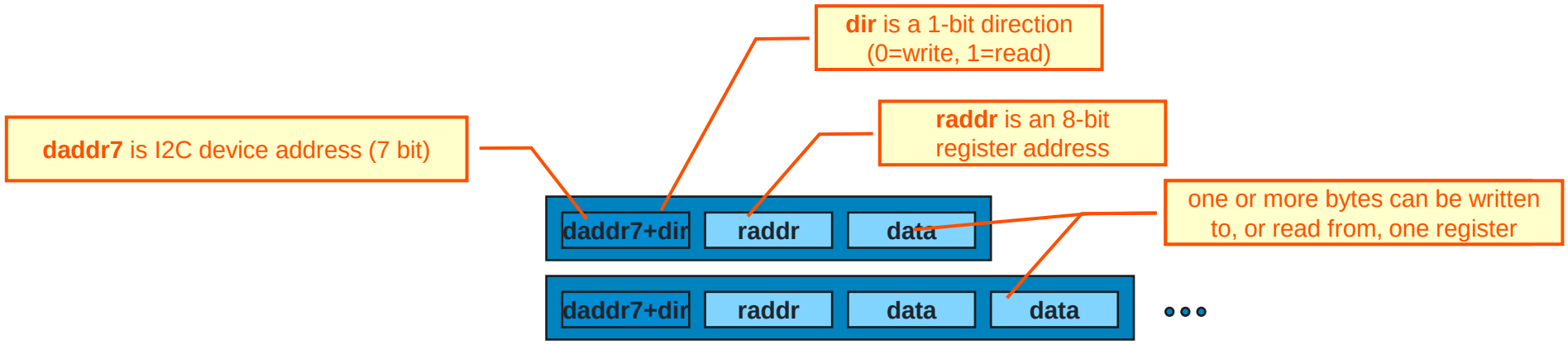
I2C protocol refresher

Register model

SAID can not
- write 0 bytes (ping)
- write 1 byte (command)
- use devices that have registers with addresses greater then 8 bit
- write to register with payload having lengths different from 1,2,4,6

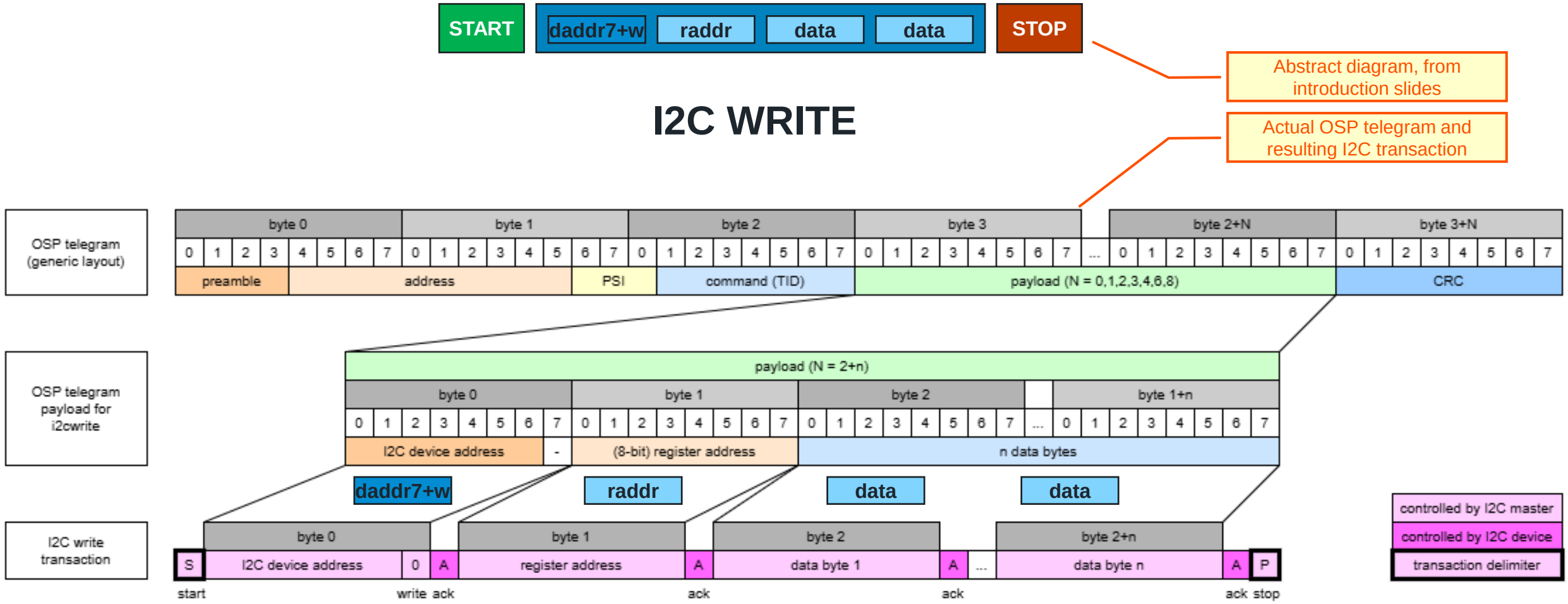
A dominant device model in I2C is the “register model”
- An I2C device models its behavior through **registers**
- Registers have an address
- By *setting* a value at that address the device is configured, or a command is given
- By *getting* a value from that address a configuration is inspected, or a state or sensor value is obtained

“Registers”
- Are not part of the I2C specification
- Used by many (but not all) I2C devices
- **Is the model implemented by the I2C master in the SAID**
- In practice, a small restriction (no 0-byte or 1-byte writes, no devices with 16 bit registers)



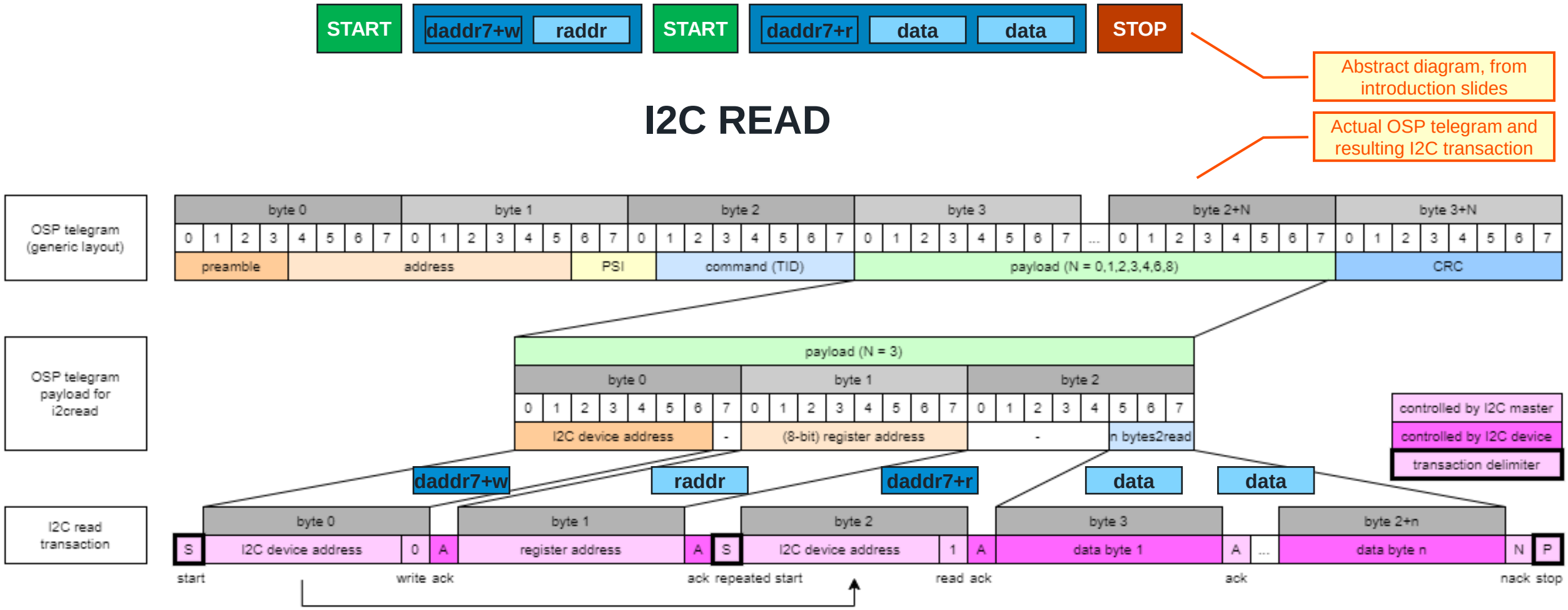
I2C in SAID

OSP telegrams wrapping I2C write transactions



I2C in SAID

OSP telegrams wrapping I2C read transactions



Libraries in detail

OSP Telegrams aoosp – exec's for I2C

The aoosp_exec module implements "macros": multiple telegrams to achieve one user action

See previous chapter

Check if the SAID at address *addr* has I2C (has the OTP bit set)

Power the I2C bus of the SAID at address *addr* (raising an error if *addr* is not a SAID or has no I2C)

```
// Sends RESET and INIT telegrams, auto detecting BiDir or Loop.
aoresult_t aoosp_exec_resetinit(uint16_t *last=0, int *loop=0);


// Reads the I2C_BRIDGE_EN bit in OTP (mirror).
aoresult_t aoosp_exec_i2cenable_get(uint16_t addr, int * enable);
// Checks if the SAID has an I2C bridge, if so, powers the I2C bus.
aoresult_t aoosp_exec_i2cpower(uint16_t addr);


// Writes to an I2C device connected to a SAID with I2C bridge..
aoresult_t aoosp_exec_i2cwrite8(uint16_t addr, uint8_t daddr7, uint8_t raddr, const uint8_t *buf, uint8_t count);
// Reads from an I2C device connected to a SAID with I2C bridge.
aoresult_t aoosp_exec_i2cread8(uint16_t addr, uint8_t daddr7, uint8_t raddr, uint8_t *buf, uint8_t count);
```

Write the *count* bytes of *buf* to register *raddr* of I2C device *daddr7* on the I2C bus of SAID *addr*

Read *count* bytes into *buf* from register *raddr* of I2C device *daddr7* on the I2C bus of SAID *addr*

Macro: polls (readi2ccfg) for completion

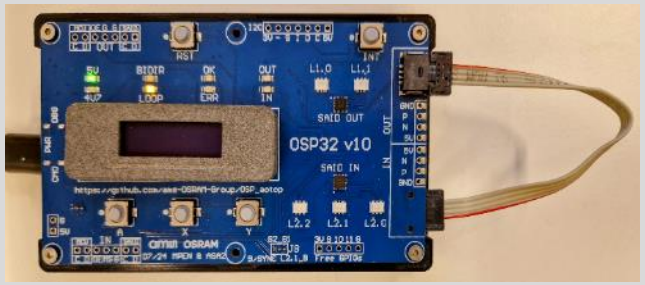
Macro: polls (readi2ccfg) for completion, then gets data (readlast)

Assignment – Training2 - eeprom

I2C EEPROM read/write

0

- From *aotop* build *training2* using “ESP32S3 Dev Module”
- Connect OSP32.OUT to OSP32.IN (Loop)
- Note SAID.OUT has an I2C EEPROM




```
void setup() {
  Serial.begin(115200);
  Serial.printf("\n\ntraining1.ino - step eeprom\n");

  aospi_init();
  aoosp_init();

  // aoosp_loglevel_set( aoosp_loglevel_tele );
  i2ceeprom();
  // Serial.printf("tx %d rx %d\n", aospi_txcount_get(), aospi_rxcount_get() );
}
```

1

The sketch should step location 40 in EEPROM by one

Fill out the gaps in the *training2* sketch

- Power the I2C bus
- Read the current register value from the EEPROM
- Show value, step value
- Write the new register value to the EEPROM

- Use the EEPROM on SAID.OUT (has I2C address 0x54)

- Open serial monitor, and press reset (or power cycle)

```
#define ADDR    0x001 // the address of the OSP node with I2C (SAID OUT on OSP32)
#define DADDR7   0x54 // I2C device address of the I2C EEPROM connected to SAID OUT
#define RADDR    0x40 // some "random" register address in the EEPROM

static void i2ceeprom( ) {
  // Reset all nodes (broadcast); all "off"; they also lose their address
  result= aoosp_exec_resetinit(&last);
  Serial.printf("resetinit() %s %d\n", aoresult_to_str(result), last );
  if( last!=2 ) Serial.printf("ERROR: unexpected topology\n");

  // Power the I2C bus
  ...

  // Read the current register value from the EEPROM
  ...

  // Show value, step value
  Serial.printf("  value %02x\n",buf[0]);
  buf[0]++;

  // Write the new register value to the EEPROM
  ...
}
```

1 Assignment

I2C EEPROM read/write


```
static void i2ceeprom( ) {
  #define    BUFSIZE 1
  uint8_t    buf[BUFSIZE];
  aoresult_t result;
  uint16_t   last;

  // Reset all nodes (broadcast); all "off"; they also lose their address
  result= aoosp_exec_resetinit(&last);
  Serial.printf("resetinit() %s %d\n", aoresult_to_str(result), last );
  if( last!=2 ) Serial.printf("ERROR: unexpected topology\n");

  // Power the I2C bus
  ...


  // Read the current register value from the EEPROM
  ...


  // Show value, step value
  Serial.printf("  value %02x\n",buf[0]);
  buf[0]++;

  // Write the new register value to the EEPROM
  ...

}
```

```
static void i2ceeprom( ) {
  #define    BUFSIZE 1
  uint8_t    buf[BUFSIZE];
  aoresult_t result;
  uint16_t   last;

  // Reset all nodes (broadcast); all "off"; they also lose their address
  result= aoosp_exec_resetinit(&last);
  Serial.printf("resetinit() %s %d\n", aoresult_to_str(result), last );
  if( last!=2 ) Serial.printf("ERROR: unexpected topology\n");

  // Power the I2C bus
  result= aoosp_exec_i2cpower(ADDR);
  Serial.printf("i2cpower(%03X) %s\n", ADDR, aoresult_to_str(result) );

  // Read the current register value from the EEPROM
  result= aoosp_exec_i2cread8(ADDR, DADDR7, RADDR, buf, BUFSIZE );
  Serial.printf("i2cread8(%03X,%02X,%02X) %s\n", ADDR, DADDR7, RADDR, aoresult_to_str(result) );

  // Show value, step value
  Serial.printf("  value %02x\n",buf[0]);
  buf[0]++;

  // Write the new register value to the EEPROM
  result= aoosp_exec_i2cwrite8(ADDR, DADDR7, RADDR, buf, BUFSIZE );
  Serial.printf("i2cwrite8(%03X,%02X,%02X) %s\n", ADDR, DADDR7, RADDR, aoresult_to_str(result) );
}
```

```
training2.ino - step eeprom
spi: init
osp: init
resetinit() ok 2
i2cpower(001) ok
i2cread8(001,54,40) ok
  value 07
i2cwrite8(001,54,40) ok
```

```
training2.ino - step eeprom
spi: init
osp: init
resetinit() ok 2
i2cpower(001) ok
i2cread8(001,54,40) ok
  value 08
i2cwrite8(001,54,40) ok
```

```
training2.ino - step eeprom
spi: init
osp: init
resetinit() ok 2
i2cpower(001) ok
i2cread8(001,54,40) ok
  value 09
i2cwrite8(001,54,40) ok
```

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

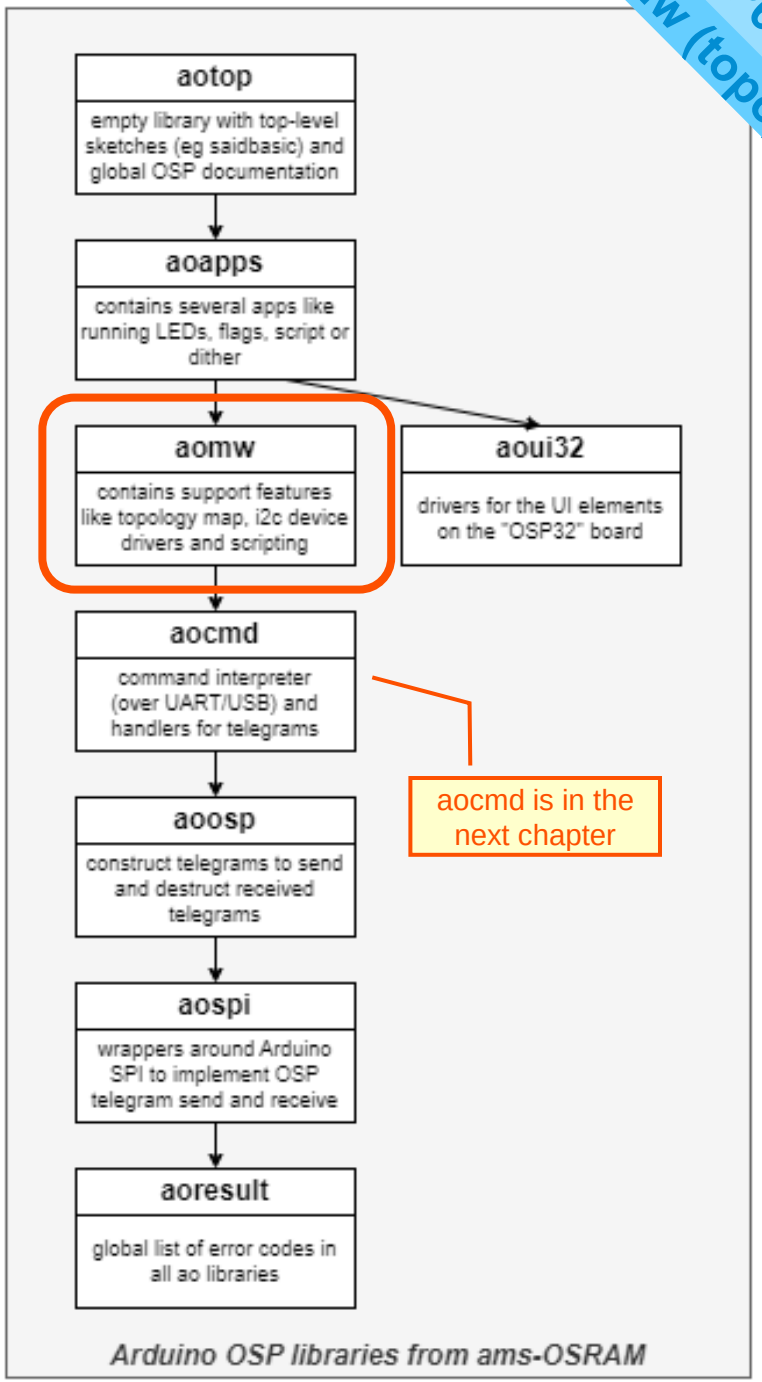
Middleware

Overview

- This library contains an assortment of software features.
  - Driver for an I2C EEPROM (on OSP32, SAIDbasic, stick)
  - Driver for an I2C I/O-expander
  - Paint (country) flags on an OSP chain (of arbitrary length)
  - Interpreter for scripted animations on an OSP chain
- Useful in making flexible demos, but are not expected in production firmware

- There is two important service: the color library and the topology manager
- This training skips the other features (read the documentation for those)
- But does explain the topology manager
- The color library is in the training appendix

- (Borrowed term from “Network topology”: the arrangement of elements in a network)



Topology manager

Overview

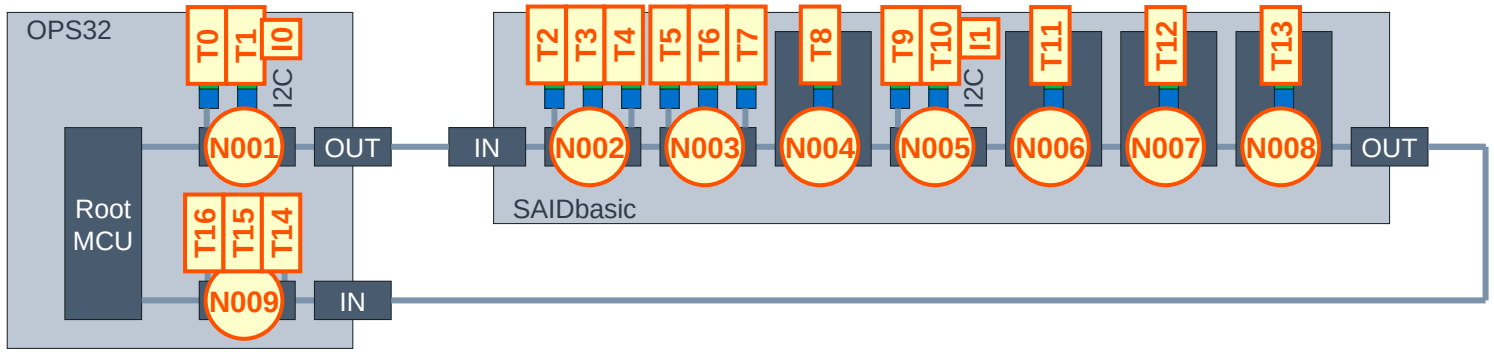
Data collected by topology manager

```
nodes(N) 1..9
triplets(T) 0..16
i2cbridges(I) 0..1
dir loop

N001 (00000040) T0 T1 I0
N002 (00000040) T2 T3 T4
N003 (00000040) T5 T6 T7
N004 (00000000) T8
N005 (00000040) T9 T10 I1
N006 (00000000) T11
N007 (00000000) T12
N008 (00000000) T13
N009 (00000040) T14 T15 T16

T0  N001.C0
T1  N001.C1
T2  N002.C0
T3  N002.C1
T4  N002.C2
T5  N003.C0
T6  N003.C1
T7  N003.C2
T8  N004
T9  N005.C0
T10 N005.C1
T11 N006
T12 N007
T13 N008
T14 N009.C0
T15 N009.C1
T16 N009.C2

I0  N001
I1  N005
```

The core task of the topology manager is

- Determine the number of nodes in the network
- For each node determine the type (RGBI, SAID)
- For each type determine “anomalies”,
  e.g. SAID with channel 2 having I2C instead of an RGB triplet
- Determine the amount and location of all RGB triplets
  (triplet means unit with red+green+blue LED)
- Determine the amount and location of all I2C bridges

- And make all this information available to the application



Topology manager

Features

The topology manager
- Does not only build the map
- Does not only offer an API to use the map
- It also prepares the OSP chain for easy use

Preparing the chain
- Reset and Init
- (Queries identity)
- Clears the error (SAIDs over-voltage)
- Enable CRC (to trap programming errors)
- Powers all I2C busses
- Sets the drive currents
- Switch to active state

The topo module abstracts away how to drive triplets
- Is the triplet on a channel
- What drive current to use
- Which telegram to send
- What PWM to send

The topo module
- defines its own dynamic range: “topo brightness range”
- ranges from 0 to 0x7FFF
- maps that to any triplet (RGBI or RGB connected to SAID)
- Driver currents configured (per node type) by topo

API
- `aomw_topo_settriplet(tix,color)`

This extra feature of “setting color of a triplet” makes it **the API** for other demo modules (no longer do they need to care which demo boards are connected).

Topology manager API

OSP Middleware aomw (aomw_topo)

Build topo map

```
// topo build in one run
aoresult_t aomw_topo_build();
```

This is what you need

Info on nodes

```
// Returns the number of nodes in the scanned chain.
uint16_t aomw_topo_numnodes();
// Returns the identity of OSP node `addr`; 1<=addr<=aomw_topo_numnodes().
uint32_t aomw_topo_node_id( uint16_t addr );
// Returns the number of triplets (RGB modules) of OSP node `addr`; 1<=addr<= aomw_topo_numnodes().
uint8_t aomw_topo_node_numtriplets( uint16_t addr );
// Returns the index of the first triplet (RGB module) driven by OSP node `addr`; 1<=addr<=aomw_topo_numnodes().
uint16_t aomw_topo_node_triplet1( uint16_t addr );
```

Info on triplets

```
// Returns the number of triplets (RGB modules) in the scanned chain.
uint16_t aomw_topo_numtriplets();
// Returns the address of the OSP node that drives triplet `tix`; 0<=tix<aomw_topo_numtriplets().
uint16_t aomw_topo_triplet_addr( uint16_t tix );
// Returns 1 if triplet `tix` is driven by an OSP node with channels; 0<=tix<aomw_topo_numtriplets().
int aomw_topo_triplet_onchan( uint16_t tix );
// Returns the channel triplet `tix` is attached to in case the triplet is driven by an OSP node with channels,
0<=tix<aomw_topo_numtriplets(). Only defined when aomw_topo_triplet_onchan(tix).
uint8_t aomw_topo_triplet_chan( uint16_t tix );
```

info on i2cbridges

```
// Returns the number of I2C bridges in the scanned chain.
uint16_t aomw_topo_numi2cbridges();
// Returns the address of the OSP node that has I2C bridge `bix`; 0<=bix<aomw_topo_numi2cbridges().
uint16_t aomw_topo_i2cbridge_addr( uint16_t bix );
```

Set color

```
// The "topo brightness range";
#define AOMW_TOPO_BRIGHTNESS_MAX 0x7FFF
// The data type
typedef struct aomw_topo_rgb_s { uint16_t r; uint16_t g; uint16_t b; const char * name; } aomw_topo_rgb_t;
// Sets the color for triplet `tix` to `rgb`
aoresult_t aomw_topo_settriplet( uint16_t tix, const aomw_topo_rgb_t*rgb );
```

Function settriplet multiplies the r,g,b values by dim/1024. This provides a global dim setting

```
// Sets the global dim-level for aomw_topo_settriplet. Function clips to 0..1024.
void aomw_topo_dim_set( int dim );
// Gets the global dim-level
int aomw_topo_dim_get();
```

Topology manager example

Switch entire OSP chain to yellow

Does all the setup details for all nodes (reset, init, clrerror, goactive, setcurchn)

```
// Build the topology map and setup the OSP chain
aoresult_t result = aomw_topo_build();
if( result!=aoresult_ok ) Serial.printf("ERROR topo_build %s\n", aoresult_to_str(result) );


// A struct with the RGB values (topo brightness range: 0000..7FFF)
aomw_topo_rgb_t rgb = { 0x1FFF, 0x1FFF, 0x0000, "yellow" };


// Loop over all triplets and set their pwm
for( int tix=0; tix<aomw_topo_numtriplets(); tix++ ) {
  result= aomw_topo_settriplet( tix, &rgb );
  if( result!=aoresult_ok ) Serial.printf("ERROR topo_setpwm(%d) %s\n", tix, aoresult_to_str(result) );
}
```

Loops over all **triplets** (not OSP nodes) setting the PWM – hiding the hardware differences

Hands-on in the next chapter

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

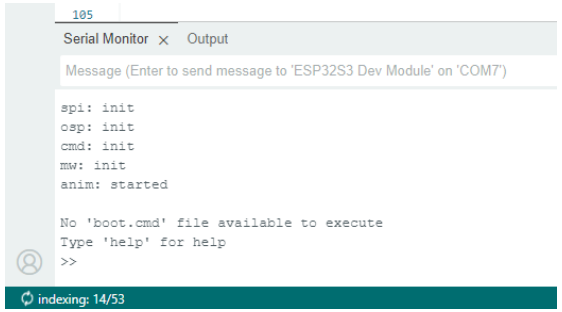
Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

Command interpreter

Overview, serial port



Serial

- The ESP32 has a Serial interface (over USB)
- Used extensively in the libraries/example to print information to PC
- This becomes visible in the “Serial Monitor” on the PC

- However, the PC can also send characters to the ESP32
- The ESP32 can interpret them and act on them

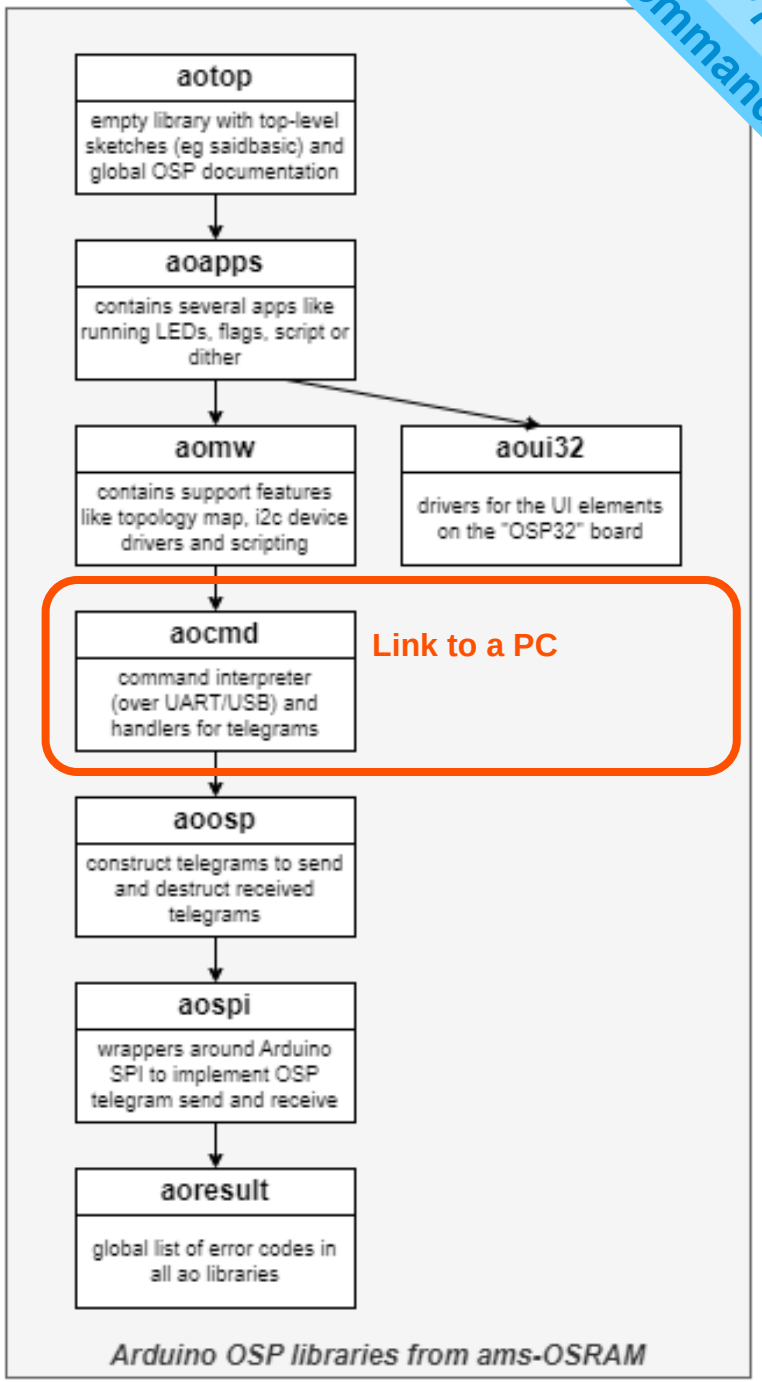
Library

- The library *aocmd* implements a command interpreter
- It is user centric (send textual commands; get textual responses)

- The library *aocmd* implements several command handlers
- For example: board, echo, file, help, osp, said, version
- Other libraries implement some extra (topo, apps)

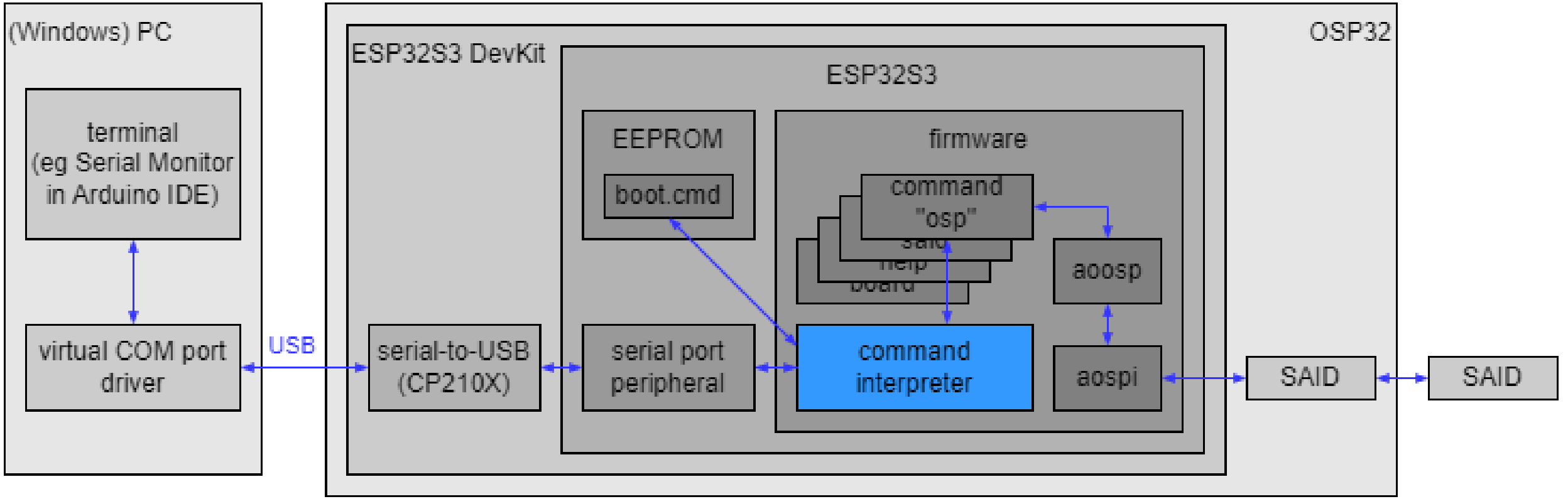
Bonus

- Bonus 1: ESP32 can store one file (boot.cmd) which is executed at start-up
- Bonus 2: There is a Python PoC (to let PC control the OSP chain)



Command interpreter

Architecture



Command interpreter

Live hands-on demonstration (sketch osplink)

Taken from the documentation https://github.com/ams-OSRAM/OSP_aocmd/blob/main/readme.md#example-commands

Generic
- banner
- prompt (>>)
- help, help echo, help echo wait
- h v, //, @

OSP
- osp enum
- osp dirmux loop
- osp info, osp info readpwmchn
- osp send
- osp tx
- wrong telegram (validate)

File
- file show
- file record
- file exec
- board reboot
- reset

Miscellaneous
- board
- echo
- version

SAID
- said i2c 000 scan
- said i2c 001 read 54 40
- said i2c 001 write 54 40 00

Topo
- topo build
- topo enum
- topo pwm 0 1FFF 0000 0000

Command interpreter

How to integrate

Upcall from version command to print app name/version

Initialize **Serial**

Register commands with the interpreter.
This is a shorthand for “register all”.

Poll **Serial**, if characters received, push them to the state machine of the command interpreter.
If a CR is received, parse and execute.

Checks if there is a boot.cmd file, if so, runs it

```
#include <aospi.h>          // aospi_init()
#include <aoosp.h>          // aoosp_init()
#include <aocmd.h>          // aocmd_register()


#define APP_NAME "Training3 – commands and topo"


// Library aocmd "upcalls" via aocmd_version_app() to print app version.
void aocmd_version_app() {
  Serial.printf("%s\n", APP_NAME );
}


void setup() {
  Serial.begin(115200); delay(1000);
  Serial.printf("\n\n%s\n",APP_NAME);

  aospi_init();
  aoosp_init();
  aocmd_init();
  aocmd_register();
  Serial.printf("\n");

  aocmd_file_bootcmd_exec_on_por();
  Serial.printf( "Type 'help' for help\n" );
  aocmd_cint_prompt();
}


void loop() {
  aocmd_cint_pollserial();
  // ... other processing ...
}
```

```
void aocmd_register() {
  aocmd_echo_register();
  aocmd_help_register();
  aocmd_version_register();
  aocmd_board_register();
  aocmd_file_register();
  aocmd_osp_register();
  aocmd_said_register();
}
```

Command interpreter

Final notes

Using the command interpreter

- Command interpreter is integrated in demos (eg saidbasic) e.g. to configure them – even via boot.cmd
- If you're not interested in the demo, but just want to test the OSP chain, use osplink sketch

Examples

- There is a template (much like on the previous slide) for using the command interpreter
- There is an example of how to make/add your own command

Make own command

- Write a handler
  ```
  void aocmd_osp_main( int argc, char * argv[] )
  ```
  using the parser functions
  ```
  aocmd_cint_parse_dec(), aocmd_cint_parse_hex(), aocmd_cint_isprefix()
  ```
- Write a help text
  ```
  const char aocmd_osp_longhelp[] = "SYNTAX: osp\n…"
  ```
- Register the new command with the command interpreter
  ```
  aocmd_cint_register(aocmd_osp_main, "osp", "sends and receives OSP telegrams", aocmd_osp_longhelp)
  ```

Typical main loop architecture

Stepping state machines of command interpreter and animation

```ino
#define APP_NAME "Training3 - cmd and topo"

void setup() {
  Serial.begin(115200); delay(1000);
  Serial.printf("\n\n%s\n",APP_NAME);

  aospi_init();
  aoosp_init();
  aocmd_init();
  aomw_init();
  aocmd_register(); // register all std cmd's
  aomw_topo_cmd_register();
  anim_start();
  Serial.printf("\n");

  aocmd_file_bootcmd_exec_on_por();
  Serial.printf( "Type 'help' for help\n" );
  aocmd_cint_prompt();
}


void loop() {
  aocmd_cint_pollserial();
  anim_step();
}
```

```ino
int      anim_ticknum;
uint32_t anim_lastms;


static void anim_start() {
  aoresult_t result = aoosp_exec_resetinit();
  if( result!=aoresult_ok ) Serial.printf("ERROR topo_build %s\n", aoresult_to_str(result) );

  result= aoosp_send_clrerror(0x000);
  if( result!=aoresult_ok ) Serial.printf("ERROR send_clrerror %s\n", aoresult_to_str(result) );

  result= aoosp_send_goactive(0x000);
  if( result!=aoresult_ok ) Serial.printf("ERROR send_goactive %s\n", aoresult_to_str(result) );

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  if( anim_ticknum%2 == 0 ) {
    aoresult_t result= aoosp_send_setpwmchn(0x001, 0, 0x1FFF/*red*/, 0x1FFF/*green*/, 0x0000/*blue*/);
    if( result!=aoresult_ok ) Serial.printf("ERROR send_setpwmchn %s\n", aoresult_to_str(result) );
  } else {
    aoresult_t result= aoosp_send_setpwmchn(0x001, 0, 0x0000/*red*/, 0x1FFF/*green*/, 0x1FFF/*blue*/);
    if( result!=aoresult_ok ) Serial.printf("ERROR send_setpwmchn %s\n", aoresult_to_str(result) );
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

Current state

Setup chain

Init state

Steady pace

Even and odd steps show yellow and cyan

Update state

Main-loop
no delay()'s to keep pollserial and step alive

ams OSRAM

Assignment – Training3 – command and topo

Assignment

0

Flash and run *training3*

1

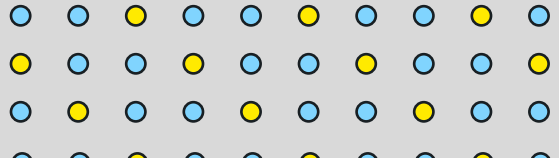
Go high-level: use **topo** in start and step

- aomw_topo_build()
- omw_topo_settriplet()

2

Code a “walking yellow animation”

- Step 1
- Step 2
- Step 3
- Step 4



3

Every step read colors from EEPROM (daddr 0x54)

- Foreground color (yellow) 6 bytes at 0x10
- Background color (cyan) 6 bytes at 0x20
- Write helper function

```
void anim_getcol(uint8_t raddr, aomw_topo_rgb_t * col )
```

```
int      anim_ticknum;
uint32_t anim_lastms;


static void anim_start() {
  aoresult_t result = aoosp_exec_resetinit();
  if( result!=aoresult_ok ) Serial.printf("ERROR topo_build %s\n", aoresult_to_str(result) );

  result= aoosp_send_clrerror(0x000);
  if( result!=aoresult_ok ) Serial.printf("ERROR send_clrerror %s\n", aoresult_to_str(result) );

  result= aoosp_send_goactive(0x000);
  if( result!=aoresult_ok ) Serial.printf("ERROR send_goactive %s\n", aoresult_to_str(result) );

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  if( anim_ticknum%2 == 0 ) {
    aoresult_t result= aoosp_send_setpwmchn(0x001, 0, 0x1FFF/*red*/, 0x1FFF/*green*/, 0x0000/*blue*/);
    if( result!=aoresult_ok ) Serial.printf("ERROR send_setpwmchn %s\n", aoresult_to_str(result) );
  } else {
    aoresult_t result= aoosp_send_setpwmchn(0x001, 0, 0x0000/*red*/, 0x1FFF/*green*/, 0x1FFF/*blue*/);
    if( result!=aoresult_ok ) Serial.printf("ERROR send_setpwmchn %s\n", aoresult_to_str(result) );
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

1 Assignment – Training3 – command and topo

Using topo


```
int      anim_ticknum;
uint32_t anim_lastms;


static void anim_start() {
  aoresult_t result = aoosp_exec_resetinit();
  if( result!=aoresult_ok ) Serial.printf(…);

  result= aoosp_send_clrerror(0x000);
  if( result!=aoresult_ok ) Serial.printf(…);

  result= aoosp_send_goactive(0x000);
  if( result!=aoresult_ok ) Serial.printf(…);

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  aoresult_t result;
  if( anim_ticknum%2 == 0 ) {
    result= aoosp_send_setpwmchn(0x001, 0, 0x1FFF, 0x1FFF, 0x0000);
    if( result!=aoresult_ok ) Serial.printf(…);
  } else {
    result= aoosp_send_setpwmchn(0x001, 0, 0x0000, 0x1FFF, 0x1FFF);
    if( result!=aoresult_ok ) Serial.printf(…);
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

```
int      anim_ticknum;
uint32_t anim_lastms;


static void anim_start() {
  aoresult_t result = aomw_topo_build();
  if( result!=aoresult_ok ) Serial.printf("ERROR topo_build %s\n", aoresult_to_str(result) );

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  aoresult_t result;
  if( anim_ticknum%2 == 0 ) {
    result= aomw_topo_settriplet( 0, &aomw_topo_yellow );
    if( result!=aoresult_ok ) Serial.printf("ERROR topo_settriplet %s\n", aoresult_to_str(result) );
  } else {
    result= aomw_topo_settriplet( 0, &aomw_topo_cyan );
    if( result!=aoresult_ok ) Serial.printf("ERROR topo_settriplet %s\n", aoresult_to_str(result) );
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

2 Assignment – Training3 – command and topo

Walking yellow


```
int      anim_ticknum;
uint32_t anim_lastms;


static void anim_start() {
  aoresult_t result = aomw_topo_build();
  if( result!=aoresult_ok ) Serial.printf(…);

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  aoresult_t result;
  if( anim_ticknum%2 == 0 ) {
    result= aomw_topo_settriplet( 0, &aomw_topo_yellow );
    if( result!=aoresult_ok ) Serial.printf(…);
  } else {
    result= aomw_topo_settriplet( 0, &aomw_topo_cyan );
    if( result!=aoresult_ok ) Serial.printf(…);
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

```
int      anim_ticknum;
uint32_t anim_lastms;


static void anim_start() {
  aoresult_t result = aomw_topo_build();
  if( result!=aoresult_ok ) Serial.printf(…);

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  for( int tix=0; tix<aomw_topo_numtriplets(); tix++ ) {
    const aomw_topo_rgb_t * col = tix%3 == anim_ticknum%3 ? &aomw_topo_yellow  : &aomw_topo_cyan ;
    aoresult_t result= aomw_topo_settriplet( tix, col );
    if( result!=aoresult_ok ) Serial.printf("ERROR settriplet(%d) %s\n", tix, aoresult_to_str(result) );
  }


  anim_ticknum++;
  anim_lastms= millis();
}
```

Math trick to determine color

ams OSRAM

3 Assignment – Training3

Read colors from EEPROM

```
#define ADDR      0x001 // the address of the OSP node with I2C (SAID OUT on OSP32)
#define DADDR7     0x54 // I2C device address of the I2C EEPROM connected to SAID OUT
#define RADDR_FG   0x10 // "random" register address in the EEPROM to store 6 bytes for fg color
#define RADDR_BG   0x20 // "random" register address in the EEPROM to store 6 bytes for bg color
#define BUFSIZE       6 // R/G/B each need two bytes
```


```
int      anim_ticknum;
uint32_t anim_lastms;




static void anim_start() {
  aoresult_t result = aomw_topo_build();
  if( result!=aoresult_ok ) Serial.printf(…);

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;




  for( int tix=0; tix<aomw_topo_numtriplets(); tix++ ) {
    const … col = … ? &aomw_topo_yellow : &aomw_topo_cyan ;
    aoresult_t result= aomw_topo_settriplet( tix, col );
    if( result!=aoresult_ok ) Serial.printf(…);
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

```
int      anim_ticknum;
uint32_t anim_lastms;

static void anim_getcol(uint8_t raddr, aomw_topo_rgb_t * col ) {
  uint8_t buf[BUFSIZE];
  // Read the color from the EEPROM from location raddr
  aoresult_t result= aoosp_exec_i2cread8(ADDR, DADDR7, raddr, buf, BUFSIZE );
  if( result!=aoresult_ok ) Serial.printf("ERROR exec_i2cread8(%02X) %s\n", raddr, aoresult_to_str(result) );
  col->r = (buf[0] & 0x7F) * 256 + buf[1];
  col->g = (buf[2] & 0x7F) * 256 + buf[3];
  col->b = (buf[4] & 0x7F) * 256 + buf[5];
}

static void anim_start() {
  aoresult_t result = aomw_topo_build();
  if( result!=aoresult_ok ) Serial.printf(…);

  anim_ticknum = 0;
  anim_lastms = millis();
  Serial.printf("anim: started\n");
}


static void anim_step() {
  if( millis()-anim_lastms < 500 ) return;

  aomw_topo_rgb_t fgcol = { 0,0,0, "fg" }; anim_getcol(RADDR_FG, &fgcol);
  aomw_topo_rgb_t bgcol = { 0,0,0, "bg" }; anim_getcol(RADDR_BG, &bgcol);

  for( int tix=0; tix<aomw_topo_numtriplets(); tix++ ) {
    const aomw_topo_rgb_t * col = tix%3 == anim_ticknum%3 ? &fgcol : &bgcol ;
    aoresult_t result= aomw_topo_settriplet( tix, col );
    if( result!=aoresult_ok ) Serial.printf("ERROR topo_setpwm(%d) %s\n", tix, aoresult_to_str(result) );
  }

  anim_ticknum++;
  anim_lastms= millis();
}
```

Assignment – Training3 – command and topo

Command

```
>> said i2c 000 scan
SAID 001 has I2C (now powered)
  00:  00  01  02  03  04  05  06  07  08  09  0a  0b  0c  0d  0e  0f
  10:  10  11  12  13  14  15  16  17  18  19  1a  1b  1c  1d  1e  1f
  20:  20  21  22  23  24  25  26  27  28  29  2a  2b  2c  2d  2e  2f
  30:  30  31  32  33  34  35  36  37  38  39  3a  3b  3c  3d  3e  3f
  40:  40  41  42  43  44  45  46  47  48  49  4a  4b  4c  4d  4e  4f
  50:  50  51  52  53 [54] 55  56  57  58  59  5a  5b  5c  5d  5e  5f
  60:  60  61  62  63  64  65  66  67  68  69  6a  6b  6c  6d  6e  6f
  70:  70  71  72  73  74  75  76  77  78  79  7a  7b  7c  7d  7e  7f
SAID 001 has 1 I2C devices

SAID 005 has I2C (now powered)
  00:  00  01  02  03  04  05  06  07  08  09  0a  0b  0c  0d  0e  0f
  10:  10  11  12  13  14  15  16  17  18  19  1a  1b  1c  1d  1e  1f
  20: [20] 21  22  23  24  25  26  27  28  29  2a  2b  2c  2d  2e  2f
  30:  30  31  32  33  34  35  36  37  38  39  3a  3b  3c  3d  3e  3f
  40:  40  41  42  43  44  45  46  47  48  49  4a  4b  4c  4d  4e  4f
  50: [50] 51  52  53  54  55  56  57  58  59  5a  5b  5c  5d  5e  5f
  60:  60  61  62  63  64  65  66  67  68  69  6a  6b  6c  6d  6e  6f
  70:  70  71  72  73  74  75  76  77  78  79  7a  7b  7c  7d  7e  7f
SAID 005 has 2 I2C devices

total 2 SAIDs have 3 I2C devices
>>
>>
>>
>> @said i2c 000 scan
[54] SAID 001 has 1 I2C devices
[20][50] SAID 005 has 2 I2C devices
total 2 SAIDs have 3 I2C devices
>>
```

```
>> said i2c 001 read 54 10 6
said(001).i2c.dev(54).reg(10) FF FF FF FF FF FF
>>
>> said i2c 001 write 54 10   1F FF   1F FF   00 00
said(001).i2c.dev(54).reg(10) 1F FF 1F FF 00 00
>>
>>
>>
>> said i2c 001 read 54 20 6
said(001).i2c.dev(54).reg(20) FF FF FF FF FF FF
>>
>> said i2c 001 write 54 20   00 00   1F FF   1F FF
said(001).i2c.dev(54).reg(20) 00 00 1F FF 1F FF
>>
>>
>> topo dim
dim 100/1024 (said 21x, rgbi 53x below max power)
>> topo dim 200
dim 200/1024 (said 11x, rgbi 28x below max power)
>>
```

Read fgcol at 10

white: (FFFF FFFF FFFF)

Write Yellow at 10

Read bgcol at 20

Write Cyan at 20

Change dim level

ams OSRAM

Part 1 – Prerequisite knowledge

Part 2 – Boards in the Arduino OSP evaluation kit

Part 3 – Libraries

Part 4 – Telegrams

Part 5 – I2C (or Telegrams part II)

Part 6 – Middleware (topo)

Part 7 – Command interpreter

Part 8 – Miscellaneous

User interface

OSP UIDriversOSP32 aoui32

Library

- This library contains drivers for the UI elements on the OSP32 board
  - A, X and Y button
  - red (error) and green (ok/heartbeat) signaling LEDs
  - the OLED screen.
- It does not depend on any of the other libraries
- The app manager uses ui32 to show the apps state

Buttons

- Buttons follow the main-loop architecture
- **`aoui32_but_scan()`** updates an internal state machine
- **`aoui32_but_wentdown()`** indicates a button press

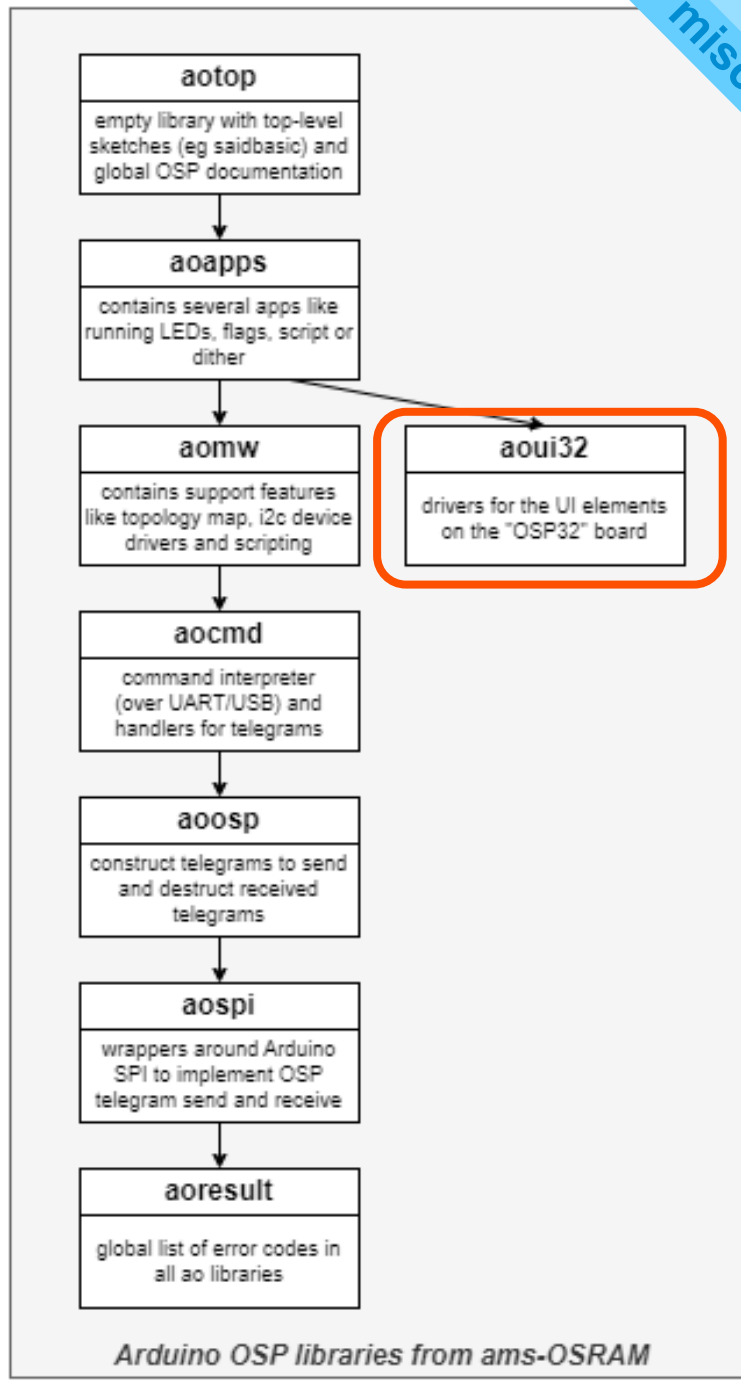
saidbasic main loop

Determine button state

Test for button transition

Update running app

```
void loop() {
  // Process incoming characters (commands)
  aocmd_cint_pollserial();

  // Check physical buttons
  aoui32_but_scan();

  // Switch to next app when A was pressed
  if( aoui32_but_wentdown(AOUI32_BUT_A) ) aoapps_mngr_switchnext();

  // Animation step in current application
  aoapps_mngr_step();
}
```



Arduino OSP libraries from ams-OSRAM

Apps

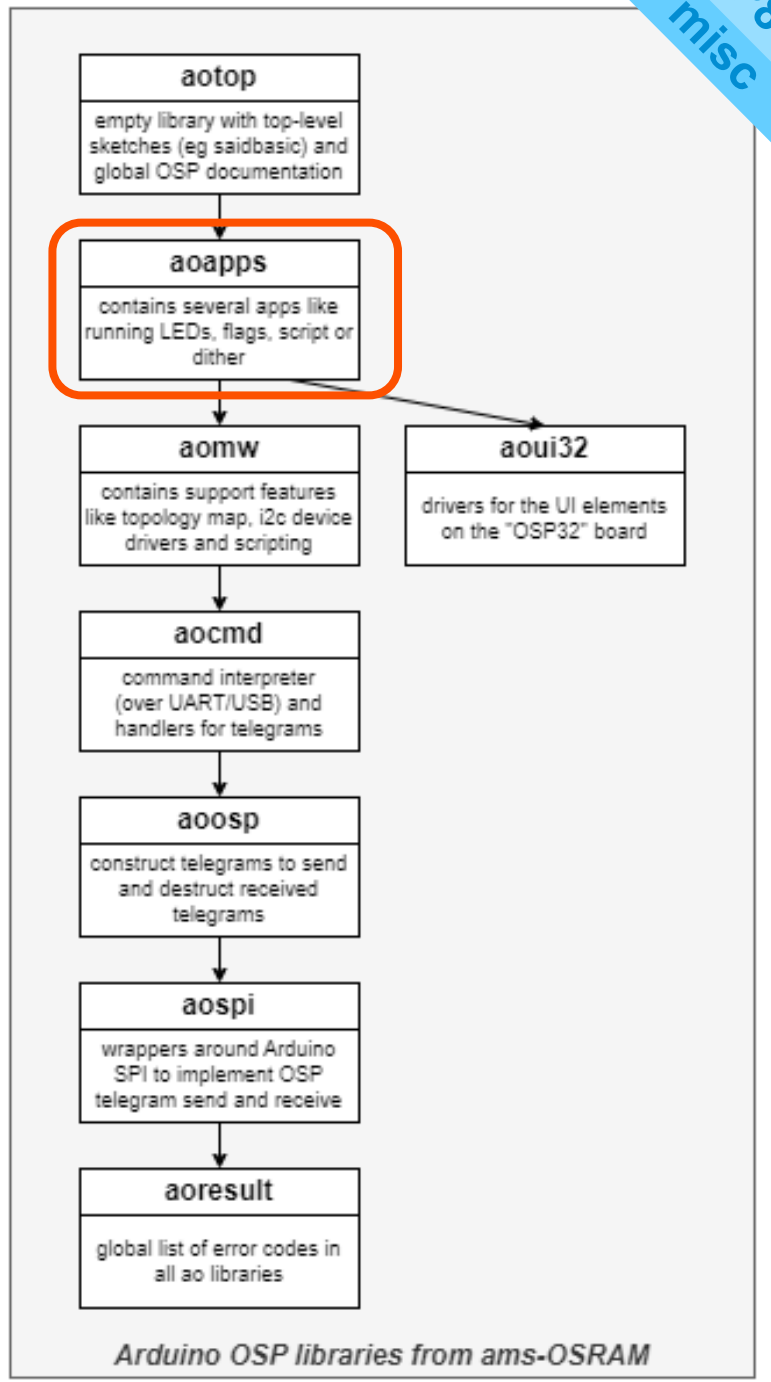
OSP ReusableApps aoapps

Main loop

- In training3 we saw an example of an animation (walking yellow lights)
- This was implemented using a state machine
- The state machine is continuously “stepped”
- The state machine compares actual time with time of previous step
- And might make a transition
- No **`delay`**’s allowed (this would kill liveliness of other state machines)
- All state machines run from the main loop – none wants delay’s

Apps

- The library aoapps contains so-called “apps”:
  running leds, switchable flag, animated script, dithering
- In this context an app is a state machine with a fixed API – start(), step(), stop()
- The library also contains an app manager
  It stops and starts apps when the A-button is pressed
- See example *saidbasic* (training0) for the apps in action



Any questions left?


ams
OSRAM