# How to Use the D-Cube Benchmarking Facility

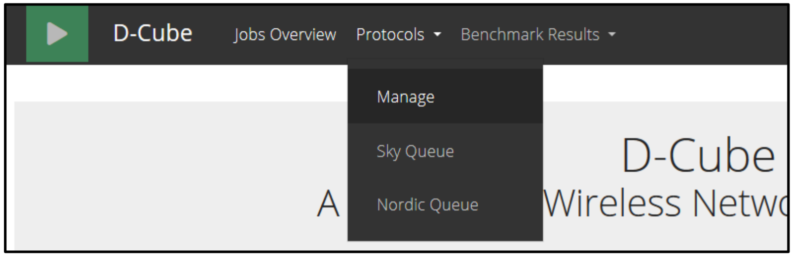
## Quick Tutorial, rev. 1

**Markus Schuß** and **Carlo Alberto Boano**
Institut für Technische Informatik
Graz University of Technology, Austria

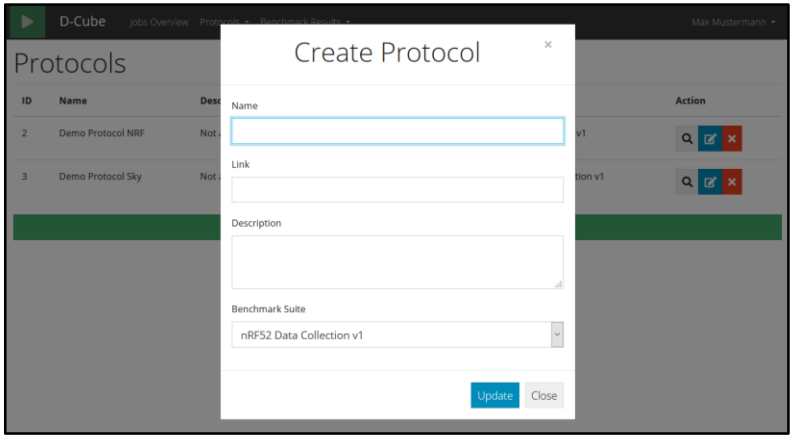
14.11.2019

# Creating a Protocol to be Benchmarked

- Use your credentials to login into D-Cube at [https://iti-testbed.tugraz.at/auth/login](https://iti-testbed.tugraz.at/auth/login)

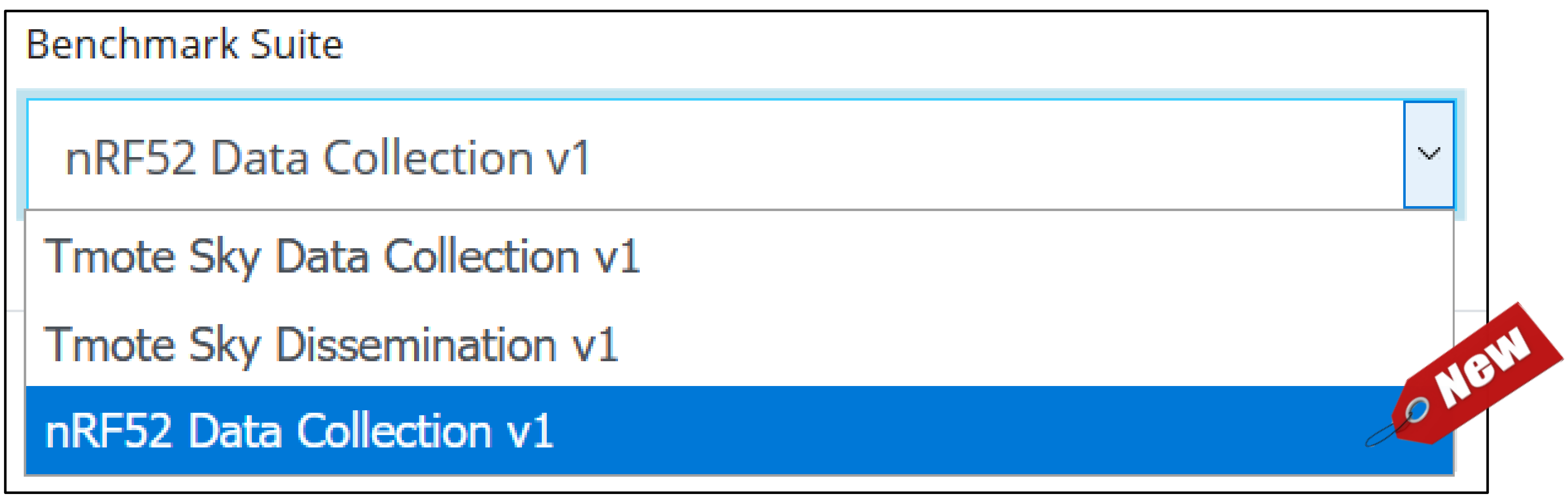



- Create a protocol under "Protocols→Manage"
  - Specify your protocol's *name*
  - Add an optional *link* (e.g., to your institution's Website, to the GIT repository containing the source code of your protocol, or to the PDF of a scientific publication describing the protocol to be tested)
  - Add a short *description* about the protocol to be tested
  - Select the *benchmark suite* on which this protocol should be tested. One protocol can be associated to a specific benchmark suite only!



# Creating a Protocol to be Benchmarked

- Available benchmark suites
  - The two Tmote Sky node suites resemble the two categories of the EWSN 2019 dependability competition (data collection and dissemination in a multi-hop network)
  - The nRF52 suite resembles a timely data collection in a multi-hop network (as specified for the EWSN 2020 dependability competition)
  - More information about the benchmark suites is available here



# Creating a Protocol to be Benchmarked

- SkyDC_1 (Tmote Sky Data Collection v1)
  SkyDD_1 (Tmote Sky Data Dissemination v1)
  - Same scenarios as EWSN Dependability Competition 2019
  - HW platform: TelosB replica / Tmote Sky
  - Performance metrics:
    1. Reliability of transmissions, i.e., the number of messages correctly reported to the intended destination(s)
    2. Average end-to-end latency in communicating each message to the intended destination(s)
    3. Average energy consumption on all nodes in the network (*)

    (*) During the first 60 seconds no data is generated and the energy consumption is not measured to allow a full bootstrap of the network
  - Application scenario:
    - Up to 8 source nodes communicating to a single destination (DC) or to a specific set of destinations (DD) in a multi-hop network
    - Source nodes generate raw sensor values of different lengths
    - No maximum per-message delay bound and out-of-order delivery possible
    - In DD, a destination cannot act as a source at the same time

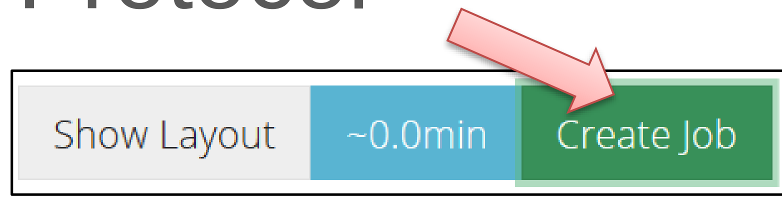
# Creating a Protocol to be Benchmarked

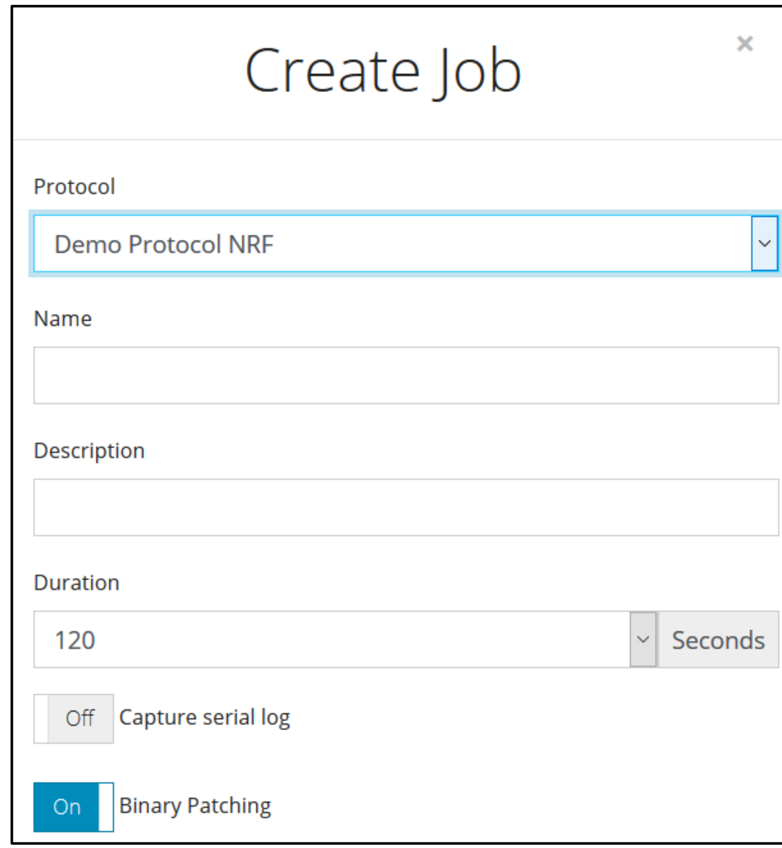
- nRFDC_1 (nRF52840 Timely Data Collection v1) New
  - HW platform: nRF52840-DK
  - Performance metrics:
    1. Reliability of transmissions, i.e., the number of messages correctly reported to each intended destination
    2. Average energy consumption on all nodes in the network (*)

    (*) During the first 60 seconds no data is generated and the energy consumption is not measured to allow a full bootstrap of the network
  - Application scenario:
    - Up to 48 source nodes generate raw sensor values of different lengths, which should be communicated to the same destination
    - The destination may be located several hops away from a source node, even when making use of the coded PHY layers available on the nRF52
    - The messages should be forwarded to the intended destination as efficiently as possible within a maximum per-message delay bound $\partial$ (i.e., the end-to-end delay of every message from its generation to its reception at the destination should be lower than $\partial$)
    - If a message has been received with an end-to-end delay greater than $\partial$, it is considered to be lost

# Start Benchmarking a Protocol

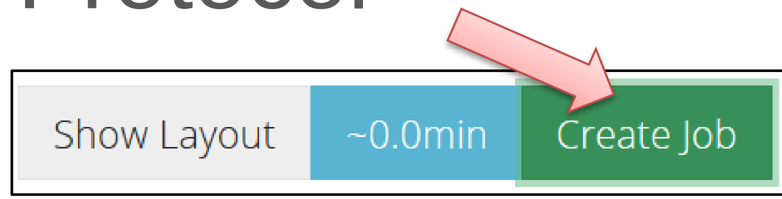
- Create Job
  - Select the protocol you created
  - Give a name to this job and a short description (e.g., testing protocol with parameter X=30)
  - Select a job duration in seconds
    <u>Note: this is currently limited to 600s!</u>
  - Specify whether to log the serial output from all nodes
    <u>Note: turning FTDI on/off severely affects the energy consumption of nodes & the accuracy of timing info!</u>
  - Specify whether to use binary patching (i.e., let the testbed inject traffic accordingly)

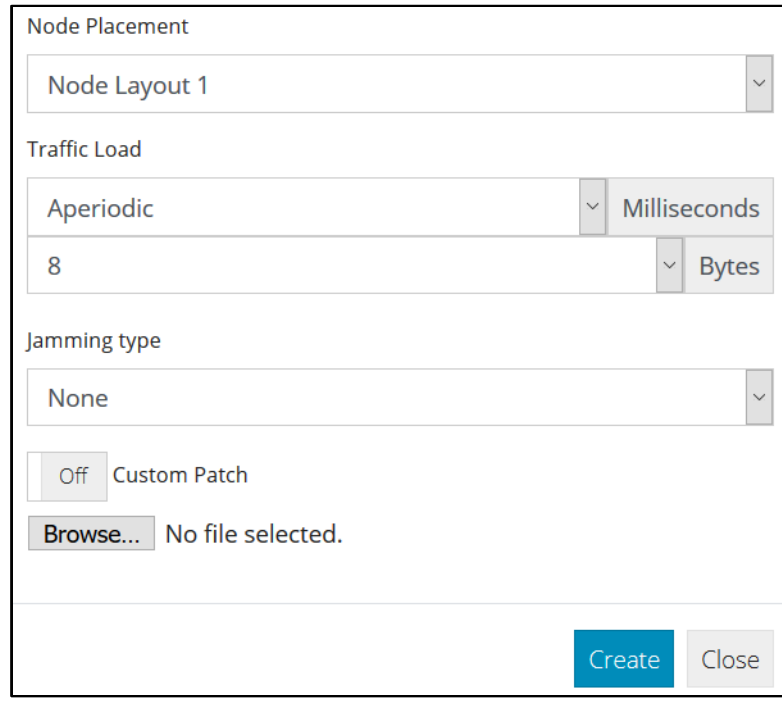


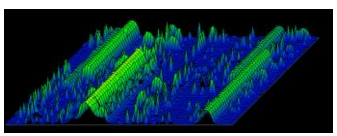



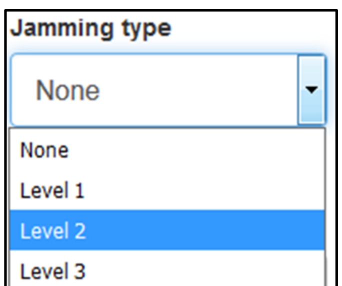
# Start Benchmarking a Protocol

- Create Job
  - Select the node placement (different node layouts are available to generalize results)
  - Select the traffic load (periodic, aperiodic)
  - Select whether radio interference should be generated ⚡
    - Select the rate at which D-Cube's observers generate Wi-Fi traffic
    - Jamming Type 1: ⚡1 only on a single frequency
    - Jamming Type 2: ⚡2 on multiple frequencies (mild)
    - Jamming Type 3: ⚡3 on multiple frequencies (stronger)
  - Select whether applying a custom patch (through XML file)
  - Upload a single binary `ihex` file to be flashed on all network nodes



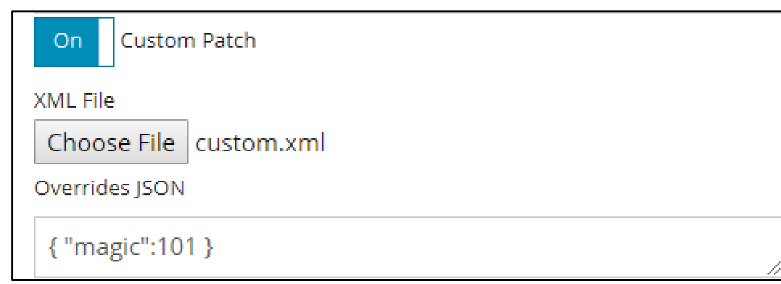





# Binary Patching Differences

- Standard Binary Patching
  - Uses the testbed’s XML
  - All values are always overwritten (0 if not used)

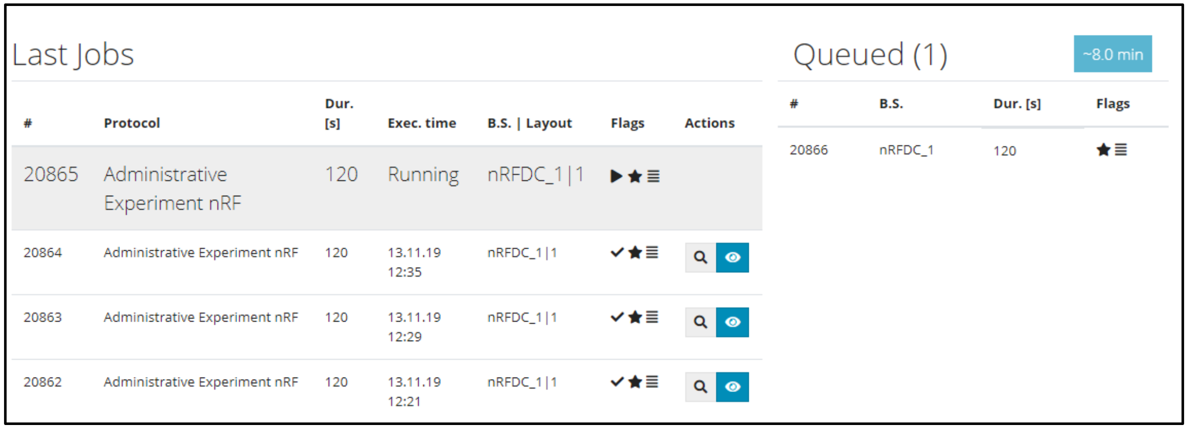

- Custom Binary Patching
  - User-provided XML (see Slide 14 for details)
  - If not specified in the “Overrides JSON” field, existing value remains




- For more info, see “Binary Patching” section later on

# Start Benchmarking a Protocol

- Flags and icons
  - Home tab shows the list of all experiments of all users (completed, running, or queued for execution)



Last Jobs

| # | Protocol | Dur. [s] | Exec. time | B.S. \| Layout | Flags | Actions |
|---|---|---|---|---|---|---|
| 20865 | Administrative Experiment nRF | 120 | Running | nRFDC_1 \| 1 | ▶ ★ ≣ | |
| 20864 | Administrative Experiment nRF | 120 | 13.11.19 12:35 | nRFDC_1 \| 1 | ✓ ★ ≣ | 🔍 👁 |
| 20863 | Administrative Experiment nRF | 120 | 13.11.19 12:29 | nRFDC_1 \| 1 | ✓ ★ ≣ | 🔍 👁 |
| 20862 | Administrative Experiment nRF | 120 | 13.11.19 12:21 | nRFDC_1 \| 1 | ✓ ★ ≣ | 🔍 👁 |

Queued (1) ~8.0 min

| # | B.S. | Dur. [s] | Flags |
|---|---|---|---|
| 20866 | nRFDC_1 | 120 | ★ ≣ |

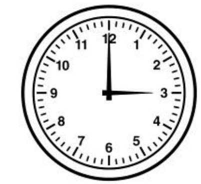
- Currently running
- Successfully completed
- Aborted or failed
- Higher priority job (testbed maintainers only)
- Log output enabled
- Visualize detailed results
- Visualize results in Grafana
- Binary patching enabled
- Custom patch added

Visible only in the per-group Sky or Nordic queue

# Testbed’s Scheduler

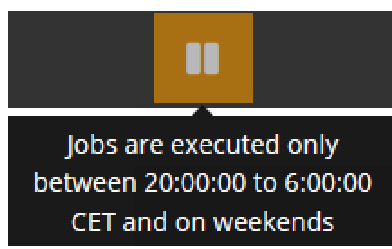
# Testbed’s Scheduler

- Jobs are typically executed between 20:00 and 6:00 CET/CEST only!
  - During weekends and Austrian public holidays, experiments are run along the 24 hours




The scheduler is currently active and processing jobs

Amount of time required by the jobs currently queued




The scheduler activity will be resumed at 20:00

Experiments can be queued anytime!

# Testbed’s Scheduler

- Jobs are typically executed between 20:00 and 6:00 CET/CEST only!
  - During weekends and Austrian public holidays, experiments are run along the 24 hours


- Why this limitation?
  - During the experiments, a harsh RF environment is created by making use of (among others) Raspberry Pi3 nodes to generate a significant amount of Wi-Fi traffic
  - When heavy Wi-Fi traffic is generated, the University’s Wi-Fi infrastructure is severely affected any can be disrupted
  - Therefore, we have agreed with TU Graz to carry out the benchmarking activities only outside the official working hours

# Testbed's Scheduler

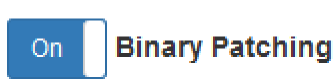
- Jobs execution policy: round-robin
  - Increases fairness in the number of experiments executed per user in a given time
  - The testbed executes jobs on a per-user and per-suite basis
    - → Scheduler iterates through the list of users
    - → For each user, it executes one job per benchmark suite (if any)
    - → If no job is scheduled, the testbed carries out the perpetual benchmarking of consolidated protocols
  - Maintenance jobs and jobs scheduled by TU Graz employees and its affiliates may have priority over other jobs

# Binary Patching

# Binary Patching

- D-Cube has the ability, to directly inject a number of input parameters into the firmware under test
  - More information available on our [CPSBench paper]

- The testbed injects the following input parameters:
  - Node ID (the ID used in the Node addresses section)
  - Traffic pattern (e.g., point-to-point traffic, multipoint-to-point traffic, …)
  - Node addresses of source and destination nodes
  - Traffic load
    - → Message length and location within the EEPROM
    - → Periodicity of messages (e.g., periodic, aperiodic, …)
    - → The deadline for messages δ after which it counts as missed




Note that you can disable binary patching when queueing your experiment for testing purposes

# Binary Patching

- Contestants need to use a pre-defined configuration struct
  - Provided in the `testbed.h` helper file
- An example on how this pre-defined configuration struct can be used is at
  - https://iti-testbed.tugraz.at/wiki/images/d/db/NRFDC_1.zip
- This example contains:
  - `testbed.h` → Helper file containing the configuration struct and some helper functions to print the values injected by the testbed
  - `custom.h & custom.xml` → Examples for custom binary patching
  - `Makefile` → Contains an example of how to configure the GCC linker's LDFLAGS correctly for binary patching
  - `Flask_placement.xml` → Contains an example of how to configure Segger Embedded Studios linker's correctly for binary patching

# Binary Patching

- Contestants need to use a pre-defined configuration struct
  - Provided in the `testbed.h` helper file
- An example on how this pre-defined configuration struct can be used is at
  - https://iti-testbed.tugraz.at/wiki/images/d/db/NRFDC_1.zip
- This example contains:
  - `main.c` → Example application built using Nordic's SDK (version 16.0)
    - → How to print values passed by the testbed (`print_testbed_config`)
    - → Read and write data to EEPROM using I2C
    - → Configure the GPIO pins and an interrupt service routine

# Binary Patching

- Your firmware application needs to include a provided header file (`testbed.h`), which contains a well-known definition of the application's input parameters
  - **The `PRINTF` macro must be defined beforehand!**

```
/* Testbed Configuration Struct Placeholder */
#define PRINTF(...) NRF_LOG_INFO(__VA_ARGS__)
#include "testbed.h"
```

- In order for the patching to work, these application input parameters need to be linked into a well-known address (`0x99000`) via the `Makefile` / `flash_placement.xml`

```
# This places the testbed configuration struct at the desired address
LDFLAGS += -Wl,--section-start -Wl,.testbedConfigSection=0x99000
LDFLAGS += -Wl,--section-start -Wl,.customConfigSection=0x98900
```

# Binary Patching

- Your firmware application needs to contain an instance of the `config_t` structure (`cfg` in the example below)
  - `cfg` enables the testbed to change several settings such as traffic pattern and traffic load before execution
  - This avoids hardcoded values in your firmware

```
//In flash configuration struct, will be replaced by binary patching
volatile config_t __attribute__((section (".testbedConfigSection"))) cfg;
```

The attribute tells the MSP430 GCC to put the variable into a new section called `testbedConfigSection` in the resulting `elf` file (`project_name.sky` in Contiki)

The `config_t` type is defined in `testbed.h`

Ensures the compiler does not remove or “optimize” the variable

# Input Parameters provided by the Testbed

- The `config_t` structure contains an array of different application input parameters (`pattern_t` struct)
- The `pattern_t` struct contains information about the delay bounds, as well as the traffic pattern and load:
  - Traffic pattern: `traffic_pattern`, `source_id[]`, `destination_id[]`
  - Traffic load: `msg_length`, `msg_offsetH`, `msg_offsetL`, `periodicity`, `aperiodic_upper_bound`, `aperiodic_lower_bound`

```c
typedef struct
{
    uint8_t traffic_pattern;                // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];         // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];    // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                     // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                    // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                    // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                   // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;         // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;         // Lower bound for aperiodic traffic in ms
    uint32_t delta;                         // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- `traffic_pattern` embeds info about the traffic pattern
  - Point-to-point (1), point-to-multipoint (2), multipoint-to-point (3), and multipoint-to-multipoint (4)
  - `traffic_pattern` is 0 if unused

```
typedef struct
{
    uint8_t traffic_pattern;                  // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];           // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];      // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                       // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                      // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                      // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                     // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;           // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;           // Lower bound for aperiodic traffic in ms
    uint32_t delta;                           // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- `source_id[TB_NUMNODES] & destination_id[TB_NUMNODES]`
  - Embed info about the address of source and destination nodes
  - Each node is identified with an 8-bit unsigned short address
    → 8-bit unsigned short value (e.g., 100, 103, 200, …)

```
typedef struct
{
    uint8_t traffic_pattern;                 // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];          // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];     // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                      // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                     // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                     // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                    // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;          // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;          // Lower bound for aperiodic traffic in ms
    uint32_t delta;                          // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- `source_id[TB_NUMNODES] & destination_id[TB_NUMNODES]`
  - Embed info about the address of source and destination nodes
  - Each node is identified with an 8-bit unsigned short address
    - → 8-bit unsigned short value (e.g., 100, 103, 200, …)
    - → We are using the node_id which is part of our config_t struct
    - → Do not rely on silicone features such as the MAC address or other UIDs as the NRF52840DKs may be replaced over time

```
typedef struct
{
    uint8_t node_id;                        // ID of the current node
    pattern_t p[TB_NUMPATTERN];             // Up to TB_NUMPATTERN parallel configurations
} config_t;
```

# Input Parameters provided by the Testbed

- traffic_pattern
  - 0: indicates that this pattern is unused and can be ignored
  - 1: only the source_id[0] and destination_id[0] are used
  - 2: the source_id[0] and all destination_id[x]!=0 are used (x = 0 … TB_NUMNODES-1)
  - 3: all source_id[x]!=0 and the destination_id[0] and are used
  - 4: all source_id[x]!=0 and destination_id[x]!=0 are used

```
typedef struct
{
    uint8_t traffic_pattern;                    // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];             // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];        // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                         // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                        // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                        // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                       // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;             // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;             // Lower bound for aperiodic traffic in ms
    uint32_t delta;                             // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- msg_length
  - Number of bytes to be written and read from EEPROM (whenever an a falling edge occurs, see later slides)
  - Messages will be at most 64 bytes

```
do{
    m_xfer_done = false;
    err_code = nrf_drv_twi_tx(&m_twi, EEPROM_ADDR, eeprom_by_address, 2, true);
    if(NRF_SUCCESS != err_code)
        continue;
    while(!m_xfer_done);
    m_xfer_done = false;
    err_code = nrf_drv_twi_rx(&m_twi, EEPROM_ADDR, val, len);
}while(NRF_SUCCESS != err_code);
while(!m_xfer_done);
```

```
typedef struct
{
    uint8_t traffic_pattern;                  // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];           // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];      // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                       // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                      // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                      // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                     // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;           // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;           // Lower bound for aperiodic traffic in ms
    uint32_t delta;                           // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- `msg_offsetH / msg_offsetL`
  - The high and low byte of the offset address in the EEPROM

```
do{
    m_xfer_done = false;
    err_code = nrf_drv_twi_tx(&m_twi, EEPROM_ADDR, eeprom_by_address, 2, true);
    if(NRF_SUCCESS != err_code)
        continue;
    while(!m_xfer_done);
    m_xfer_done = false;
    err_code = nrf_drv_twi_rx(&m_twi, EEPROM_ADDR, val, len);
}while(NRF_SUCCESS != err_code);
while(!m_xfer_done);
```

Selective Read in which we load the address of the memory location in the EEPROM where to read/write afterwards (16 bits), see https://www.onsemi.com/pub/Collateral/CAT24M01-D.PDF

```
/* Common addresses definition for eeprom. */
#define EEPROM_ADDR          (0xA0U >> 1)
```

- `0x50` is the 7-bit device address of EEPROM
- Selecting the memory location is always a write operation (even when reading from it)

```
typedef struct
{
    uint8_t traffic_pattern;                  // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];           // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];      // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                       // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                      // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                      // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                     // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;           // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;           // Lower bound for aperiodic traffic in ms
    uint32_t delta;                           // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- `periodicity`
  - Contains the period in milliseconds at which new messages are provided in EEPROM (signaled via a GPIO falling edge event, see later slides)
  - A value of 0 for periodicity indicates aperiodic traffic
    - → For aperiodic traffic, one can use the `aperiodic_upper_bound` and `aperiodic_lower_bound` bounds
    - → New messages will be provided at random times between these two bounds
    - → Both bounds are in milliseconds

```
typedef struct
{
    uint8_t traffic_pattern;                 // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];          // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];     // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                      // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                     // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                     // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                    // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;          // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;          // Lower bound for aperiodic traffic in ms
    uint32_t delta;                          // The delay bound delta in ms
} pattern_t;
```

# Input Parameters provided by the Testbed

- `delta`
  - Contains the deadline in milliseconds after which a new message is considered missed, even if it was to be delivered afterwards
  - A value of 0 for delta indicates that no such deadline exists
  - `delta` is only available in the nRFDC_1 benchmark suite

```
typedef struct
{
    uint8_t traffic_pattern;              // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
    uint8_t source_id[TB_NUMNODES];       // Only source_id[0] is used for p2p/p2mp
    uint8_t destination_id[TB_NUMNODES];  // Only destination_id[0] is used for p2p/mp2p
    uint8_t msg_length;                   // Message length in bytes in/to EEPROM
    uint8_t msg_offsetH;                  // Message offset in bytes in EEPROM (high byte)
    uint8_t msg_offsetL;                  // Message offset in bytes in EEPROM (low byte)

    uint32_t periodicity;                 // Period in ms (0 indicates aperiodic traffic)
    uint32_t aperiodic_upper_bound;       // Upper bound for aperiodic traffic in ms
    uint32_t aperiodic_lower_bound;       // Lower bound for aperiodic traffic in ms
    uint32_t delta;                       // The delay bound delta in ms
} pattern_t;
```

# Multiple Patterns

- The `config_t` structure contains an array of different application input parameters (`pattern_t` struct)
  - More than one `pattern_t` can be active at the same time, depending on the benchmark suite
    → With `TB_NUMPATTERN = 8`, we have up to `p[0]` … `p[7]`
  - The node_id is shared across all `pattern_t`
    → `node_id` is only in the nRFDC_1 benchmark suite

```
typedef struct
{
    uint8_t node_id;                    // ID of the current node
    pattern_t p[TB_NUMPATTERN];         // Up to TB_NUMPATTERN parallel configurations
} config_t;
```

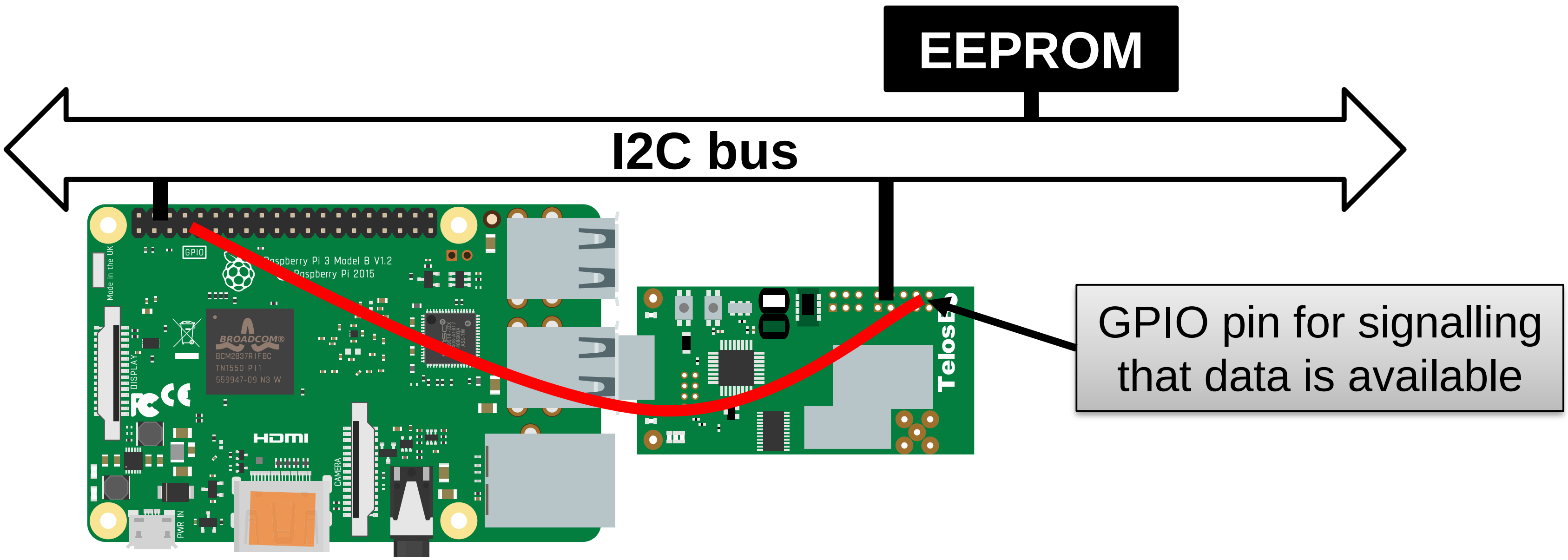
# Printing Helper Function

- `print_testbed_config`
  - The `testbed.h` file also provides a function to print the input parameters injected by the testbed
  - You can use this function during the first experiments to make sure that your code works as expected
  - Make sure to enable the logging of serial output in the testbed (see later slide)

```
void
print_testbed_config(config_t* cfg)
{
    printf("Testbed configuration:\n");
    uint8_t i;
    for(i=0;i<TB_NUMPATTERN;i++)
```

# EEPROM Communication

- Messages to be sent over the network are available in an EEPROM connected via I2C bus
  - MR44V100A FeRAM (“EEPROM”) (located at address `0x50` on the bus), datasheet: http://www.lapis-semi.com/en/data/datasheet-file_db/Memory/FEDR44V100A-01.pdf
  - The I2C bus is shared between the testbed’s observer module (Raspberry Pi 3) and the target node (Tmote Sky or nRF52840)
  - A pre-defined GPIO pin is used to signal that data is available



# EEPROM Communication

- Messages to be sent over the network are available in an EEPROM connected via I2C bus
  - In most benchmark suites, no messages are generated in the first 60 seconds (setup time)
    - → Allows routing-based solutions to establish trees and discover parents
    - → Energy consumed during this time is not considered for the final metric
    - → The initial setup time is not necessarily interference-free
  - A pre-defined GPIO pin is used to signal that data is available
    - → The GPIO used corresponds to Pin 24 n the Rapsberry Pi<br>e.g. Pin P2.6 (GIO3) on a TelosB Sky or Pin P1.02 (D1) on a nRF52840DK
    - → Same Pin is used for both source and destination nodes

- On the falling edge the testbed's observer module (Raspberry Pi 3) will try to read the EEPROM
- Make sure the I2C bus has been freed by this point!
- Latency measurement is carried out between falling edges

# EEPROM Communication

- The I2C bus is shared between the observer module (testbed's RPi3) and the target node (Tmote Sky / nRF52840)
  - I2C does not really support multi-master without arbitration
  - We use a single GPIO pin (RPI Pin 24) to indicate read or write operations
  - Keep in mind that read and write operations take time!

    → Do not write more than once every 20ms to give the observer module time to read the content

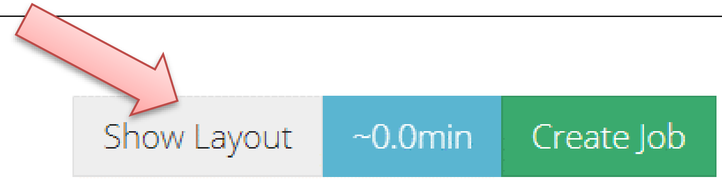
      → You can also watch the I2C clock (SCL) for activity to ensure data that has been read

# EEPROM Communication

- Once a falling edge is detected on the source, the data can be read and transmitted to the destination
- Once the destination receives the message, it actuates the GPIO to high, reads the data, and lowers the GPIO

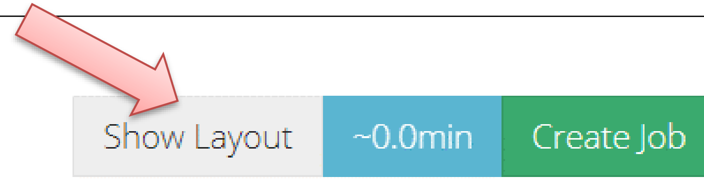

Note that, the rising edge on the GPIO of a source node is not necessarily constant, as the EEPROM may skew the clock

# Layout of Nodes

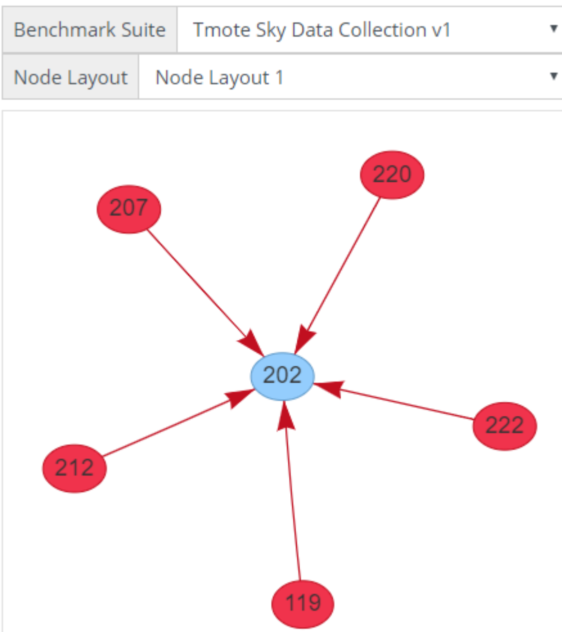



- For each benchmark suite, different node layouts are available
  - Different configurations or sets of source and destination nodes
  - Binary patched by the testbed when running a job
  - Node layout can be specified when creating a job
  - For most Benchmarks
    - → Layout 1 is average (typically multi-hop) node placement
    - → Layout 2 is a simple node layout with nodes reachable within a single hop for initial tests
    - → Layout "Empty Configuration" is intended to use with binary patching disabled for testing purposes
      - ❖ No data is generated on the EEPROM

# Layout of Nodes

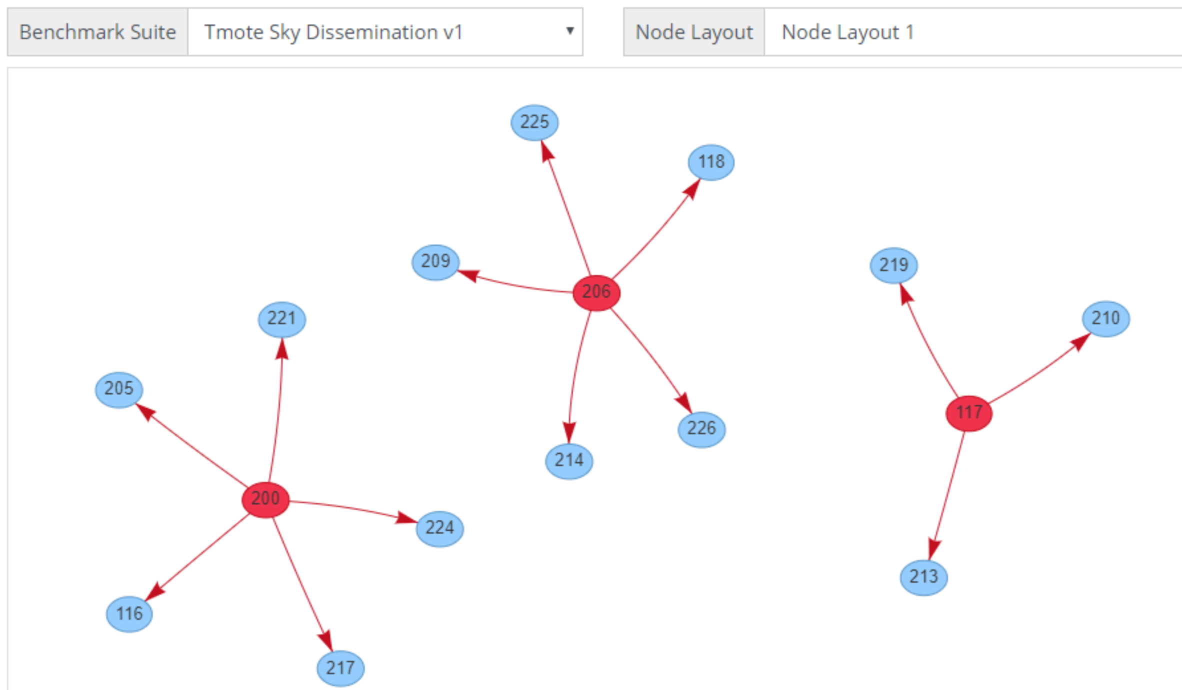



- How to interpret the available node layouts




- 207, 220, 212, 119, and 222 are source nodes
- 202 is the destination node

(These nodes are not necessarily in range)




Three source nodes (200, 206, and 117) transmitting to several destination nodes

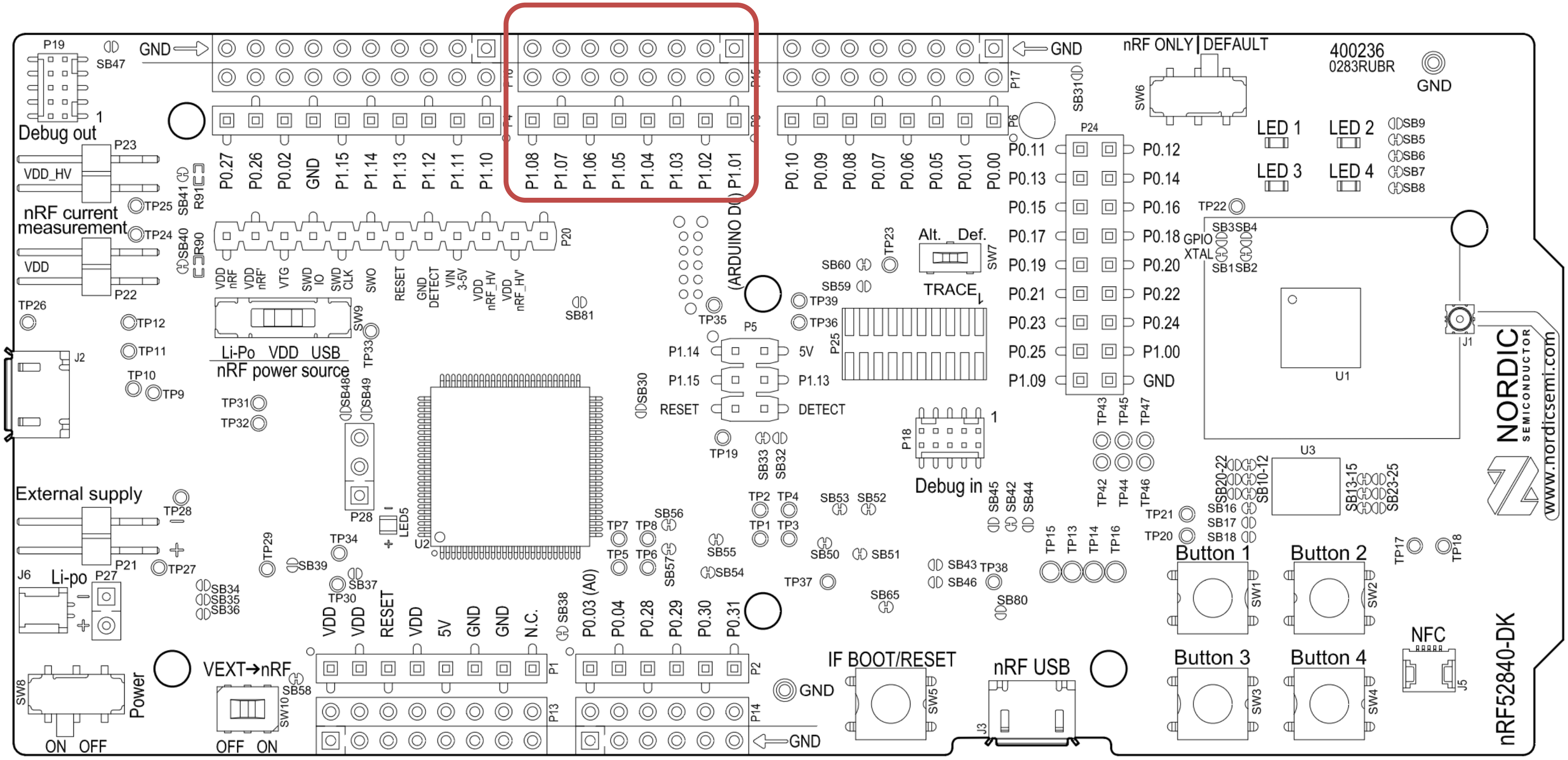
(These nodes are not necessarily in range)

# Hardware Details

(nRF52840DK)

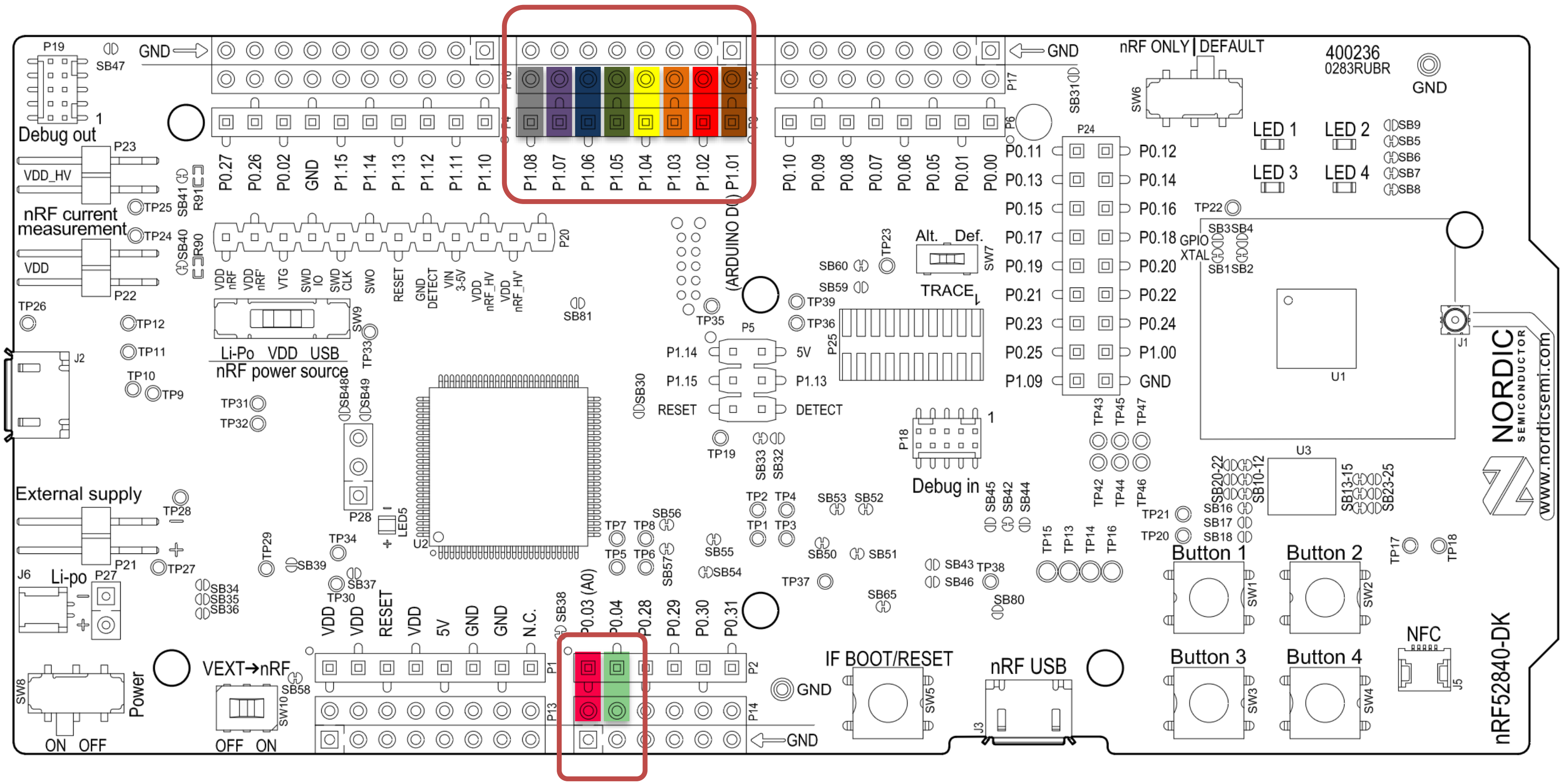
# GPIO Pins

- The testbed facility is connected to eight of the pins available in the Arduino-compatible connectors



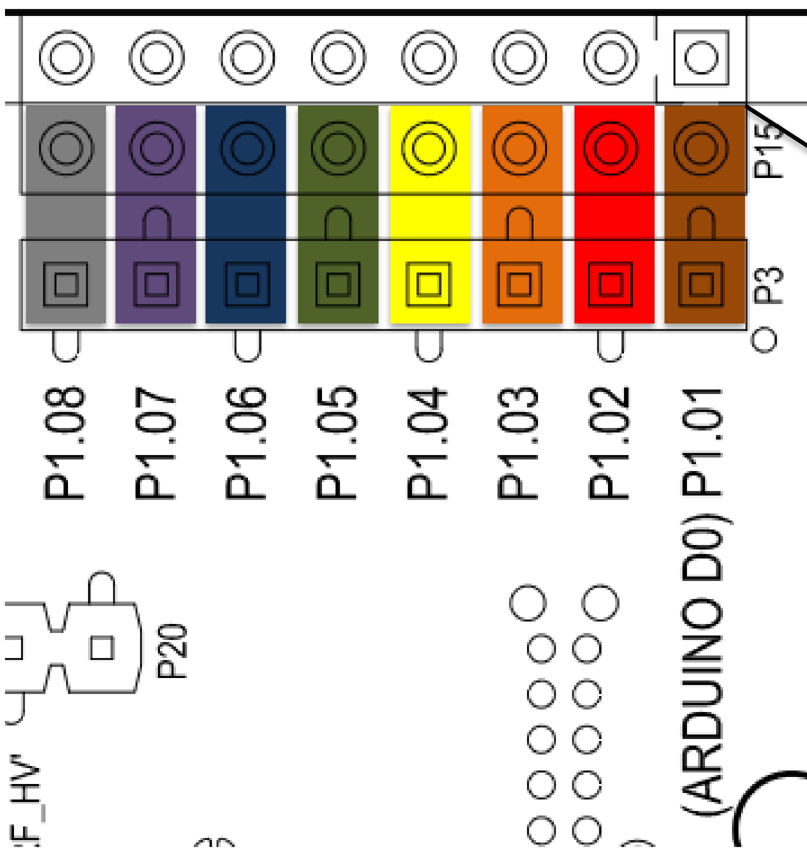
# GPIO Pins

- The testbed facility is connected to eight of the pins available in the Arduino-compatible connectors



# GPIO Pins

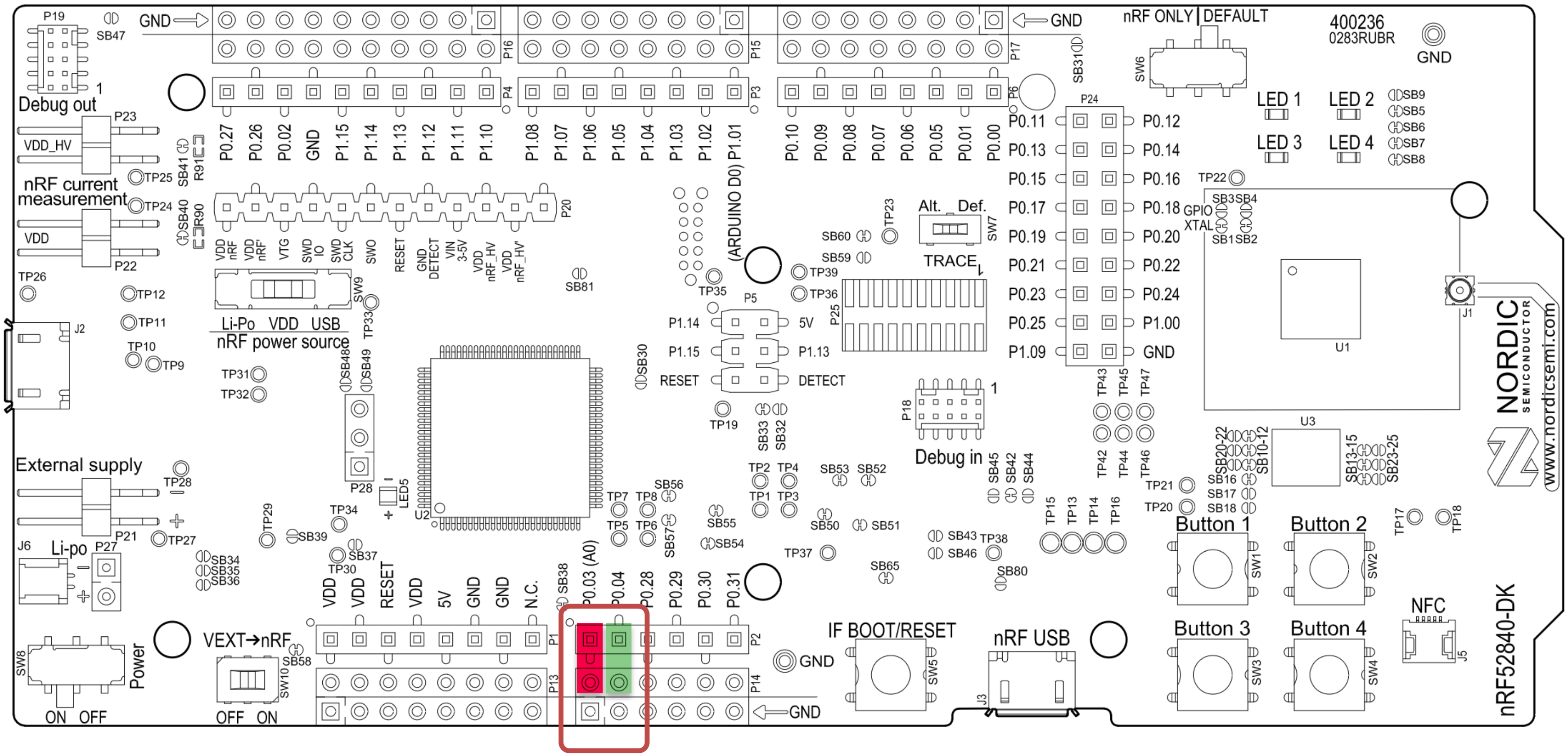
- The testbed facility is connected to eight of the pins available in the Arduino-compatible connectors




| Sensor Node | Grafana |
|---|---|
| P1.01/D0 | GPIO 17 |
| P1.02/D1 | GPIO 4 |
| P1.03/D2 | GPIO 18 |
| P1.04/D3 | GPIO 27 |
| P1.05/D4 | GPIO 22 |
| P1.06/D5 | GPIO 23 |
| P1.07/D6 | GPIO 24 |
| P1.08/D7 | GPIO 25 |

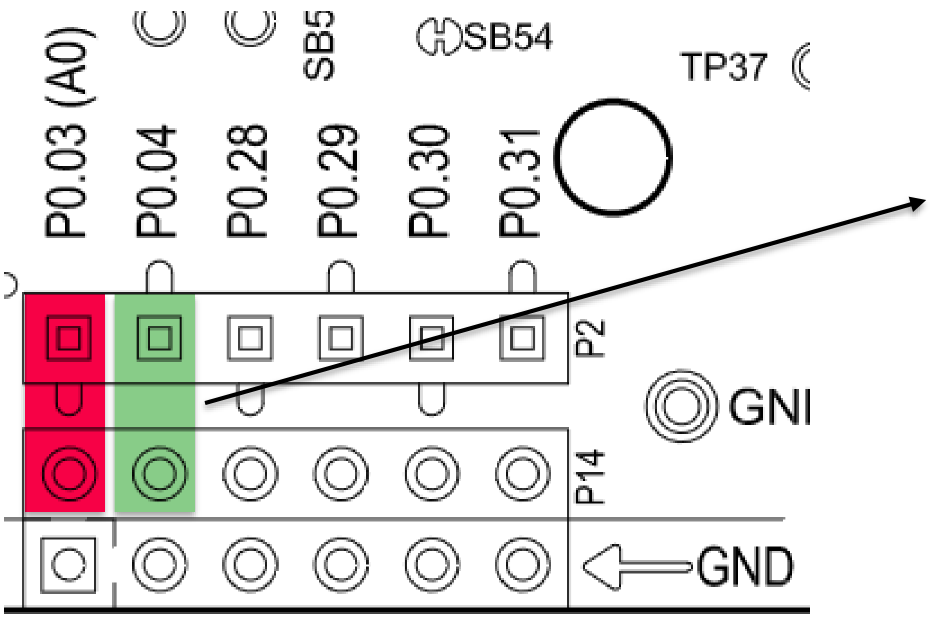
# I2C Pins

- The EEPROM is connected to two of the pins available in the Arduino-compatible connectors



# I2C Pins

- The EEPROM is connected to two of the pins available in the Arduino-compatible connectors

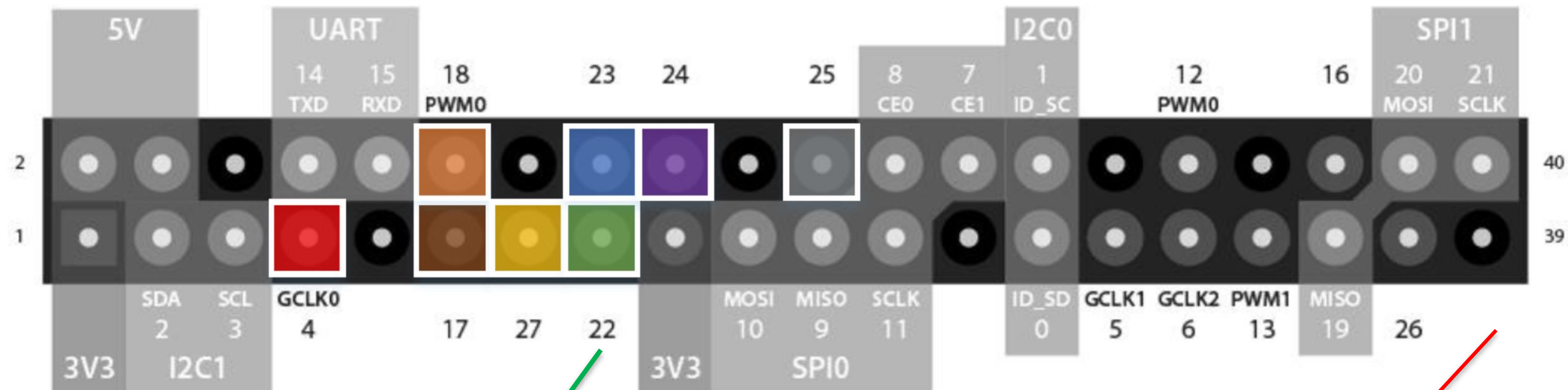



| Sensor Node | I2C |
|---|---|
| P0.03/A0 | SCL |
| P0.04/A1 | SDA |

# Numbering of GPIO Pins in Grafana

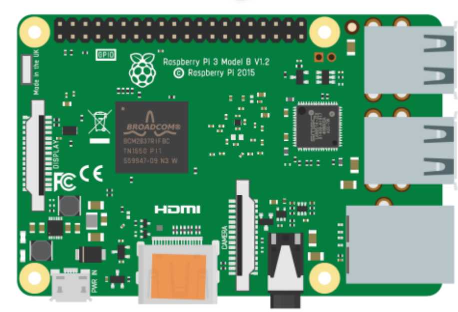
- The GPIO numbers in Grafana correspond to the GPIO pin number to which the sensor node testbed is attached on D-Cube's Observer (Raspberry Pi3)

Raspberry Pi GPIO BCM numbering




(Pin 22 on Raspberry Pi3)

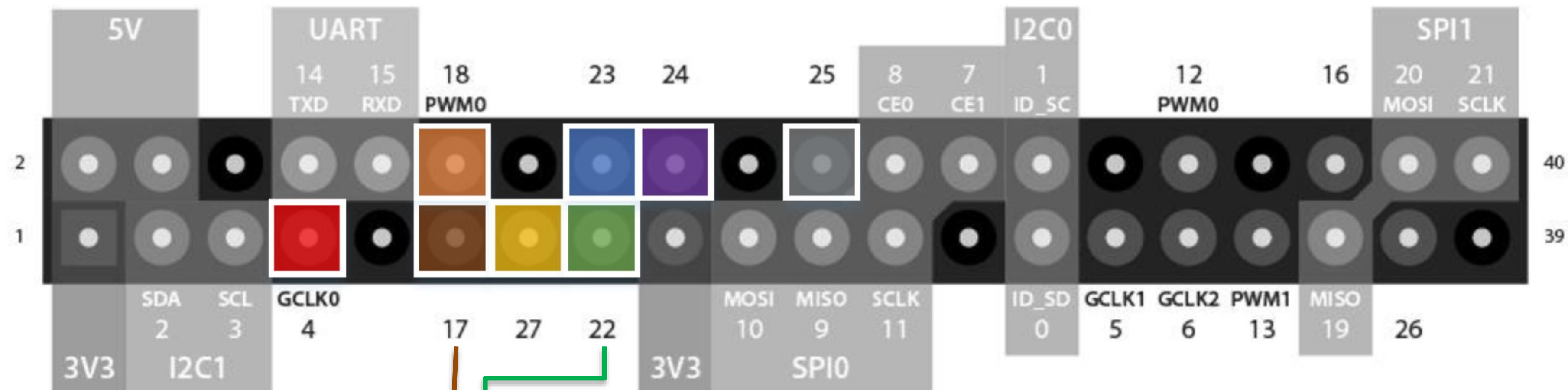
| | |
|---|---|
| **P1.05/D4** | **GPIO 22** |



# Numbering of GPIO Pins in Grafana

- The GPIO numbers in Grafana correspond to the GPIO pin number to which the sensor node testbed is attached on D-Cube’s Observer (Raspberry Pi3)

**Raspberry Pi GPIO** BCM numbering




| Sensor Node | Grafana |
|---|---|
| P1.01/D0 | GPIO 17 |
| P1.02/D1 | GPIO 4 |
| P1.03/D2 | GPIO 18 |

| Sensor Node | Grafana |
|---|---|
| P1.04/D3 | GPIO 27 |
| P1.05/D4 | GPIO 22 |
| P1.06/D5 | GPIO 23 |

| Sensor Node | Grafana |
|---|---|
| P1.07/D6 | GPIO 24 |
| P1.08/D7 | GPIO 25 |

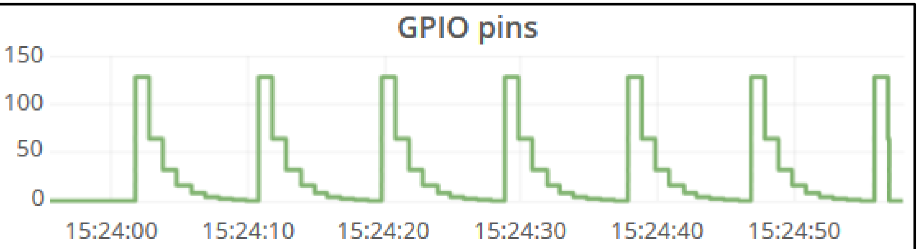
# GPIO Pins in Grafana

- In the “Overview of individual nodes” tab, the displayed “GPIO pins” numbers in Grafana are derived with the following mapping:

- Example: “GPIO pins” value of 18
  - 18 = 0001 0010 in binary
  - Using Grafana’s mapping:
  - P1.01=0; P1.02=1; P1.03=0; P1.04=0;
  - P1.05=1; P1.06=0; P1.07=0; P1.08=0;

```
gpio=0;
gpio=gpioRead(17);
gpio=(gpio<<1) | gpioRead(4);
gpio=(gpio<<1) | gpioRead(18);
gpio=(gpio<<1) | gpioRead(27);
gpio=(gpio<<1) | gpioRead(22);
gpio=(gpio<<1) | gpioRead(23);
gpio=(gpio<<1) | gpioRead(24);
gpio=(gpio<<1) | gpioRead(25);
```

**Mapping in Grafana**

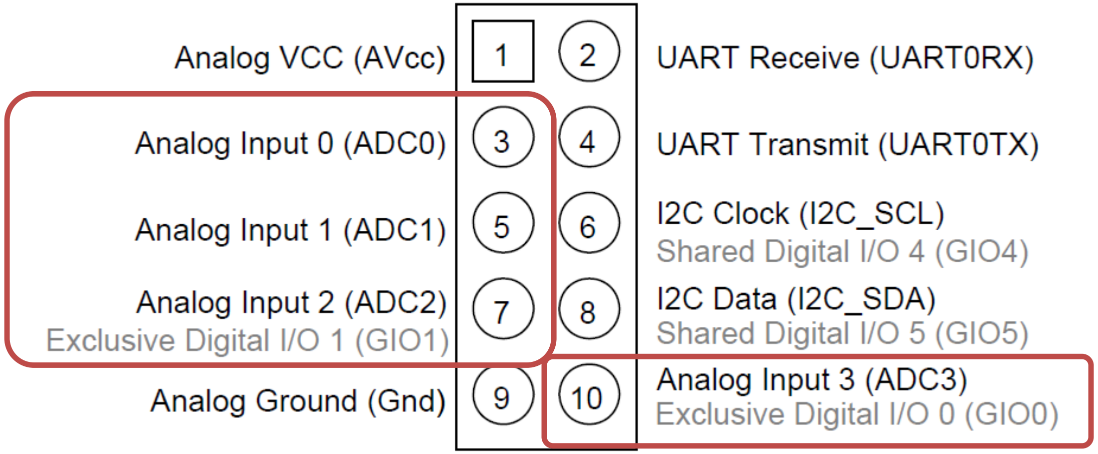


# Hardware Details

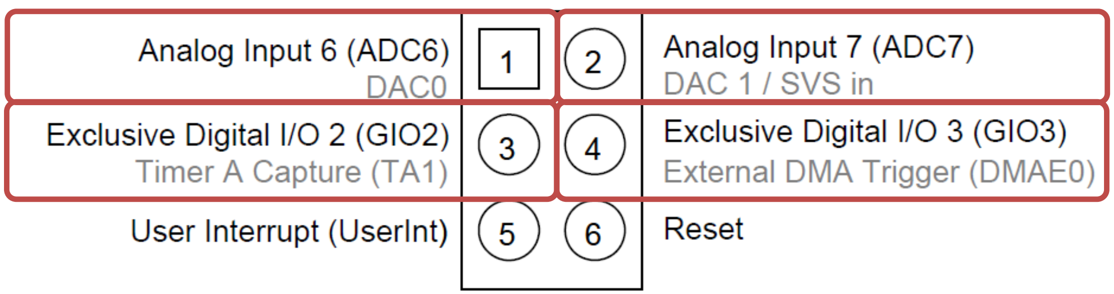
## TelosB Sky

# GPIO Pins

- The testbed facility is connected to eight of the pins available in the 10-pin and 6-pin expansion connector




**10-pin expansion connector (U2)**

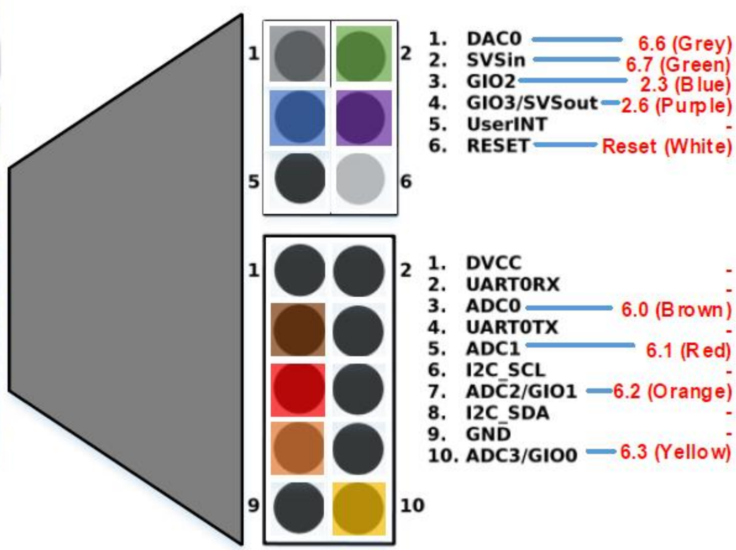



**6-pin expansion connector (U28)**

# GPIO Pins

- The testbed facility is connected to eight of the pins available in the 10-pin and 6-pin expansion connector

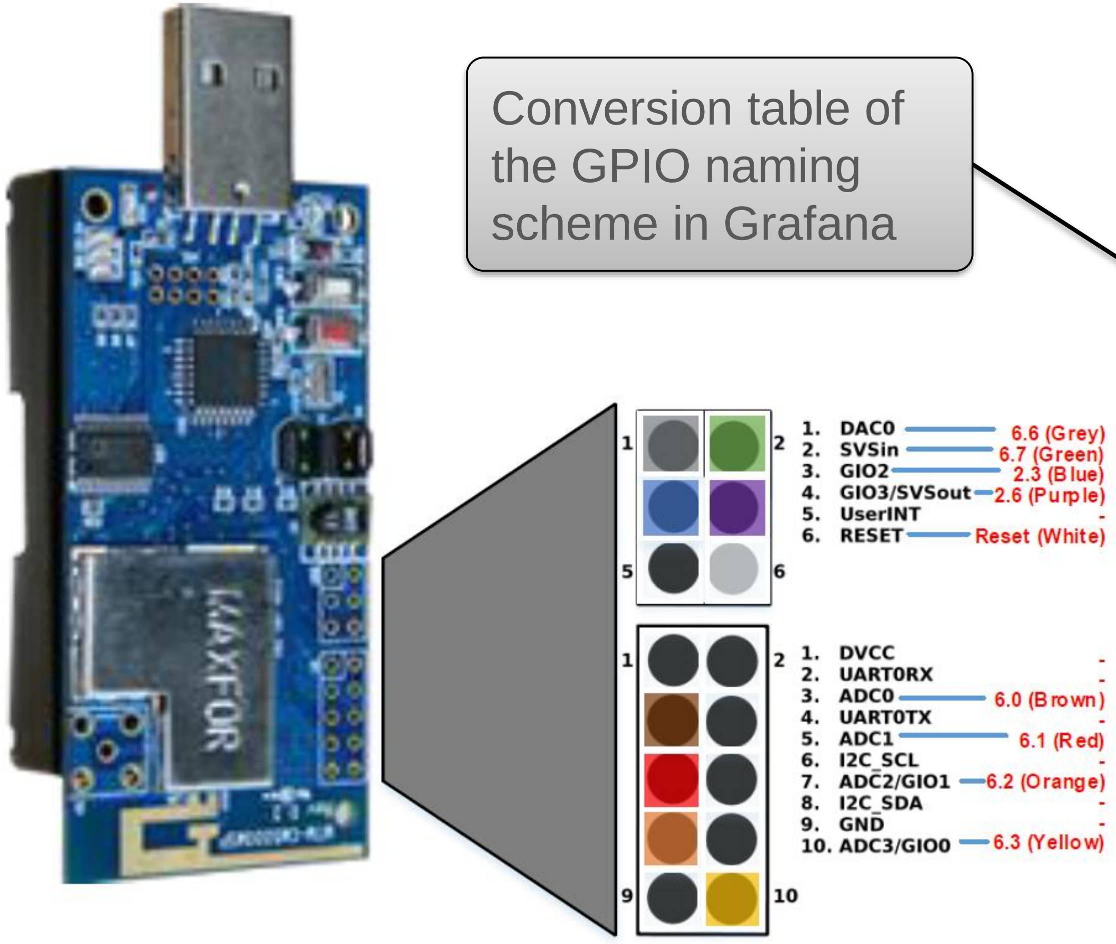
Example on how to configure the pins of the sensor node




```
//ADC0                    //ADC7
P6SEL &= ~(BIT0);         P6SEL &= ~(BIT7);
P6DIR |= BIT0;            P6DIR |= BIT7;

//ADC1                    //GIO2
P6SEL &= ~(BIT1);         P2SEL &= ~(BIT3);
P6DIR |= BIT1;            P2DIR |= BIT3;

//ADC2                    //GIO3
P6SEL &= ~(BIT2);         P2SEL &= ~(BIT6);
P6DIR |= BIT2;            P2DIR |= BIT6;

//ADC3                    //ADC6 / DAC0
P6SEL &= ~(BIT3);         P6SEL &= ~(BIT6);
P6DIR |= BIT3;            P6DIR |= BIT6;
```

# GPIO Pins

- The testbed facility is connected to eight of the pins available in the 10-pin and 6-pin expansion connector



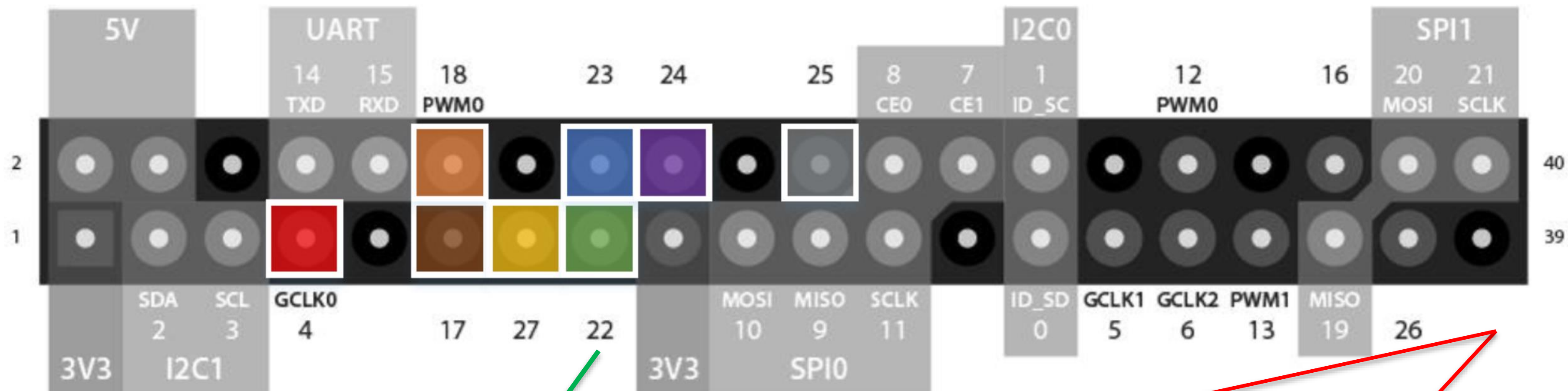

Conversion table of the GPIO naming scheme in Grafana

| Sensor Node | Grafana |
|---|---|
| ADC0 | GPIO 17 |
| ADC1 | GPIO 4 |
| ADC2/GIO1 | GPIO 18 |
| ADC3/GIO0 | GPIO 27 |
| ADC7/SVSin | GPIO 22 |
| GIO2 | GPIO 23 |
| GIO3/SVSout | GPIO 24 |
| DAC0/ADC6 | GPIO 25 |

# Numbering of GPIO Pins in Grafana

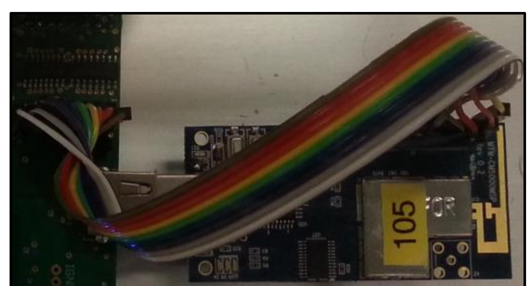
- The GPIO numbers in Grafana correspond to the GPIO pin number to which the sensor node testbed is attached on D-Cube's Observer (Raspberry Pi3)

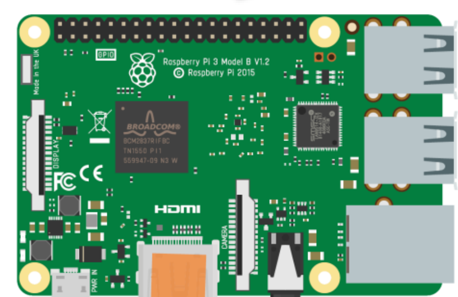
**Raspberry Pi GPIO BCM numbering**




(Pin 22 on Raspberry Pi3)

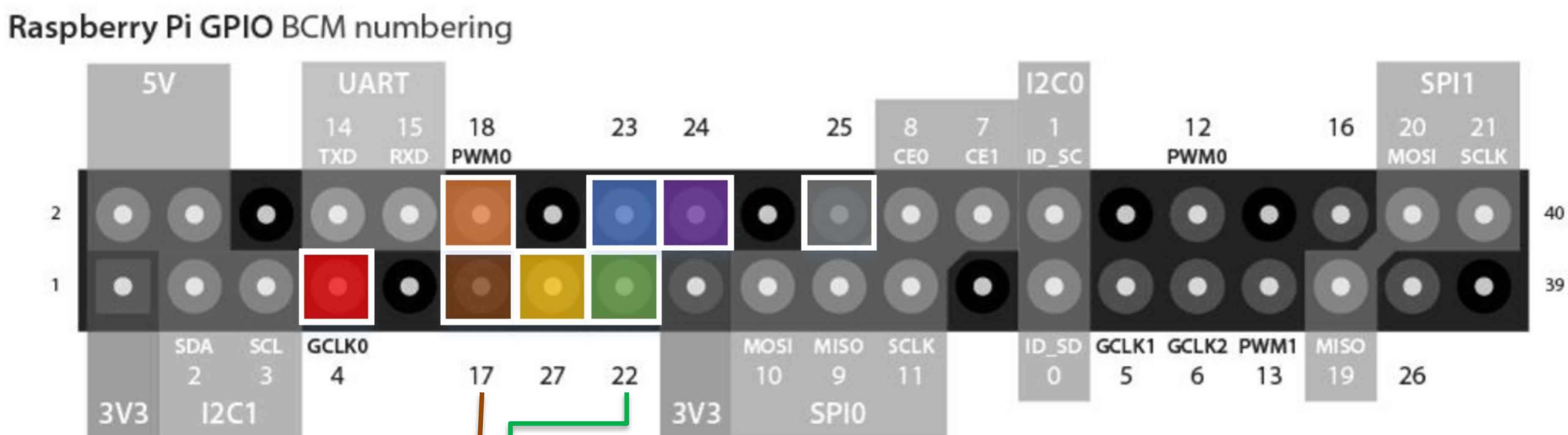
| ADC7/SVSin | GPIO 22 |
|---|---|



# Numbering of GPIO Pins in Grafana

- The GPIO numbers in Grafana correspond to the GPIO pin number to which the sensor node testbed is attached on D-Cube's Observer (Raspberry Pi3)




| Sensor Node | Grafana |
|---|---|
| ADC0 | GPIO 17 |
| ADC1 | GPIO 4 |
| ADC2/GIO1 | GPIO 18 |

| Sensor Node | Grafana |
|---|---|
| ADC3/GIO0 | GPIO 27 |
| ADC7/SVSin | GPIO 22 |
| GIO2 | GPIO 23 |

| Sensor Node | Grafana |
|---|---|
| GIO3/SVSout | GPIO 24 |
| DAC0/ADC6 | GPIO 25 |

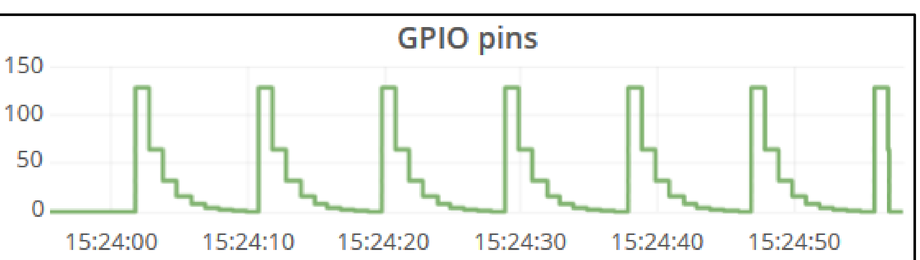
# GPIO Pins in Grafana

- In the “Overview of individual nodes” tab, the displayed “GPIO pins” numbers in Grafana are derived with the following mapping:

- Example: “GPIO pins” value of 18
  - 18 = 0001 0010 in binary
  - Using Grafana’s mapping:
  - ADC0=0; ADC1=0; ADC2=0; ADC3=1
  - SVSin=0; GIO2=0; GIO3=1; ADC6=0

```
gpio=0;
gpio=gpioRead(17);
gpio=(gpio<<1) | gpioRead(4);
gpio=(gpio<<1) | gpioRead(18);
gpio=(gpio<<1) | gpioRead(27);
gpio=(gpio<<1) | gpioRead(22);
gpio=(gpio<<1) | gpioRead(23);
gpio=(gpio<<1) | gpioRead(24);
gpio=(gpio<<1) | gpioRead(25);
```

**Mapping in Grafana**



# Contact

- dcube@iti.tugraz.at

- D-Cube team
  - Markus Schuss
    - E-mail: markus.schuss@tugraz.at
    - Tel.: +43 316 873 6403
  - Carlo Alberto Boano
    - E-mail: cboano@tugraz.at
    - Tel.: +43 316 873 6413