

How to Use the D-Cube Benchmarking Facility

Quick Tutorial, rev. 1

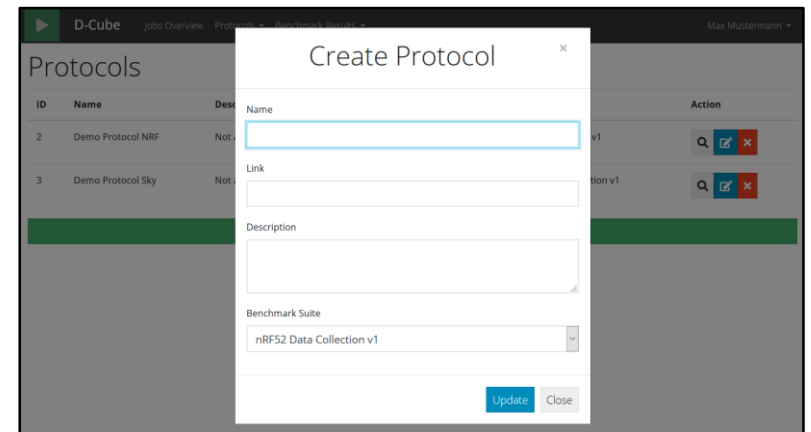
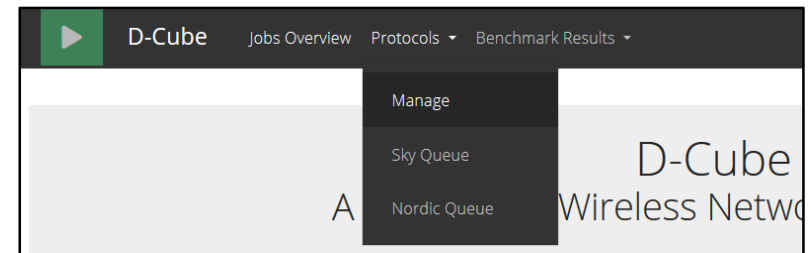
Markus Schuß and **Carlo Alberto Boano**

Institut für Technische Informatik
Graz University of Technology, Austria

28.11.2019

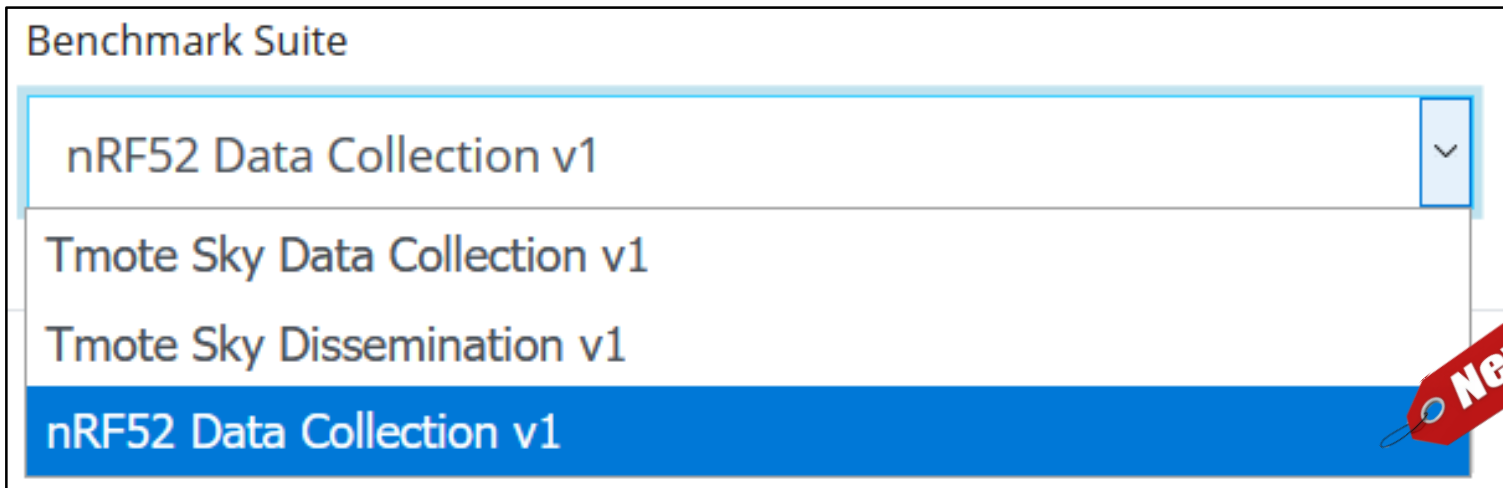
Creating a Protocol to be Benchmarked

- Use your credentials to login into D-Cube at <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) 
 - 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 ∂ (i.e., the end-to-end delay of every message from its generation to its reception at the destination should be lower than ∂)
 - If a message has been received with an end-to-end delay greater than ∂ , 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
Note: this is currently limited to 600s!
- Specify whether to log the serial output from all nodes 
Note: turning FTDI on/off severely affects the energy consumption of nodes & the accuracy of timing info!
- Specify whether to use binary patching (i.e., let the testbed inject traffic accordingly) 



Show Layout

~0.0min

Create Job

Create Job

Protocol

Demo Protocol NRF

Name

Description

Duration

120

Seconds

Off

Capture serial log

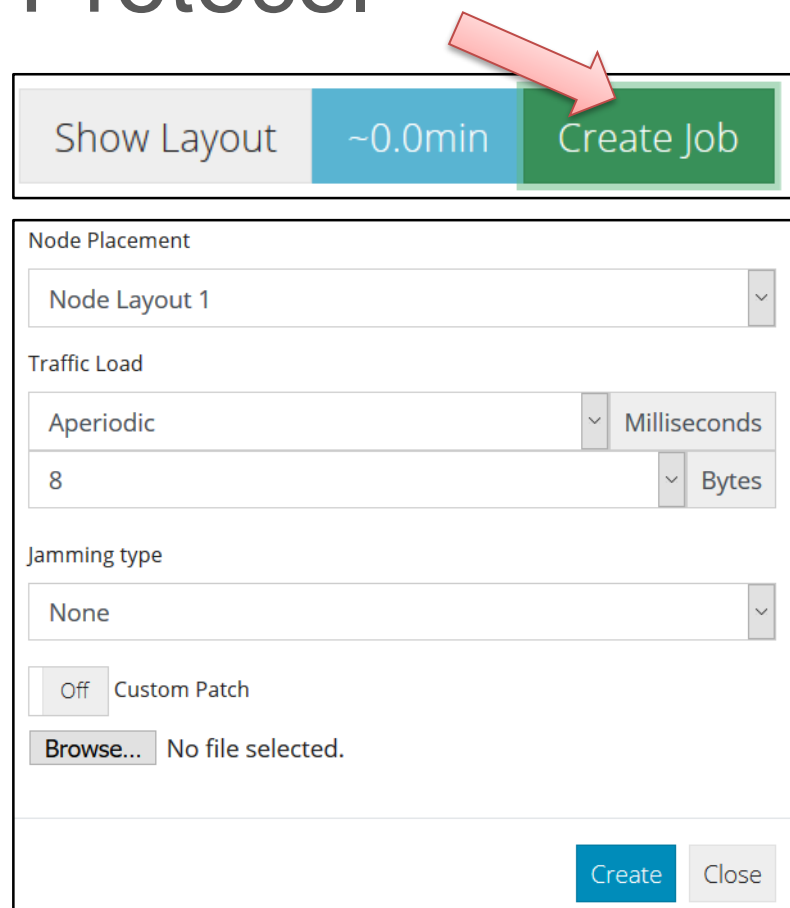
On

Binary Patching

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



Show Layout ~0.0min **Create Job**

Node Placement

Node Layout 1

Traffic Load

Aperiodic Milliseconds

8 Bytes

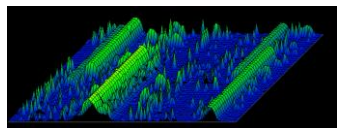
Jamming type

None

Off Custom Patch

Browse... No file selected.

Create Close



Jamming type

None

None

Level 1

Level 2

Level 3

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

☒ Binary Patching

Node Placement

Node Layout 1

Traffic Load

5000

Milliseconds

8

Bytes

Jamming type

None

☒ Custom Patch

XML File

Choose File custom.xml

Overrides JSON

{ "magic":101 }

- 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							Queued (1) ~8.0 min			
#	Protocol	Dur. [s]	Exec. time	B.S. Layout	Flags	Actions	#	B.S.	Dur. [s]	Flags
20865	Administrative Experiment nRF	120	Running	nRFDC_1 1	▶ ★ ≡		20866	nRFDC_1	120	★ ≡
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	✓ ★ ≡	🔍 👁				



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

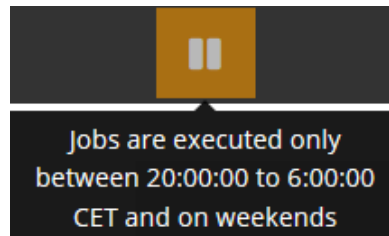
Testbed's Scheduler

- 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



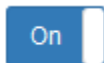
Testbed's Scheduler

- 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

**Binary Patching**

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
- Each benchmark suite has its own struct that can be downloaded at https://iti-testbed.tugraz.at/wiki/index.php/Overview_of_Available_Suites
- An **example** on how this pre-defined configuration struct can be used is available [here](#) and 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 in the `????` for binary patching
 - `Flask_placement.xml` → Contains an example of how to configure Segger Embedded Studios linker's correctly in the `????` for binary patching

Binary Patching

- Contestants need to use a pre-defined configuration struct
 - Provided in the `testbed.h` helper file
- Each benchmark suite has its own struct that can be downloaded at https://iti-testbed.tugraz.at/wiki/index.php/Overview_of_Available_Suites
- An **example** on how this pre-defined configuration struct can be used is available [here](#) and 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!**

```
63  /* Testbed Configuration Struct Placeholder */
64  #define PRINTF(...) NRF_LOG_INFO(__VA_ARGS__)
65  #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`

```
122 # This places the testbed configuration struct at the desired address
123 LDFLAGS += -Wl,--section-start -Wl,.testbedConfigSection=0x99000
124 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

```
20  
21 //In flash configuration struct, will be replaced by binary patching  
22 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`

```

7  typedef struct
8  {
9      uint8_t traffic_pattern;           // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10     uint8_t source_id[TB_NUMNODES];    // Only source_id[0] is used for p2p/p2mp
11     uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12     uint8_t msg_length;                 // Message length in bytes in/to EEPROM
13     uint8_t msg_offsetH;                 // Message offset in bytes in EEPROM (high byte)
14     uint8_t msg_offsetL;                 // Message offset in bytes in EEPROM (low byte)
15
16     uint32_t periodicity;                 // Period in ms (0 indicates aperiodic traffic)
17     uint32_t aperiodic_upper_bound;      // Upper bound for aperiodic traffic in ms
18     uint32_t aperiodic_lower_bound;      // Lower bound for aperiodic traffic in ms
19     uint32_t delta;                      // The delay bound delta in ms
20 } 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

```

7  typedef struct
8  {
9      uint8_t traffic_pattern;           // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10     uint8_t source_id[TB_NUMNODES];    // Only source_id[0] is used for p2p/p2mp
11     uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12     uint8_t msg_length;                 // Message length in bytes in/to EEPROM
13     uint8_t msg_offsetH;                 // Message offset in bytes in EEPROM (high byte)
14     uint8_t msg_offsetL;                 // Message offset in bytes in EEPROM (low byte)
15
16     uint32_t periodicity;                 // Period in ms (0 indicates aperiodic traffic)
17     uint32_t aperiodic_upper_bound;      // Upper bound for aperiodic traffic in ms
18     uint32_t aperiodic_lower_bound;      // Lower bound for aperiodic traffic in ms
19     uint32_t delta;                       // The delay bound delta in ms
20 } 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, ...)

```

7  typedef struct
8  {
9      uint8_t traffic_pattern;           // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10     uint8_t source_id[TB_NUMNODES];   // Only source_id[0] is used for p2p/p2mp
11     uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12     uint8_t msg_length;                // Message length in bytes in/to EEPROM
13     uint8_t msg_offsetH;               // Message offset in bytes in EEPROM (high byte)
14     uint8_t msg_offsetL;               // Message offset in bytes in EEPROM (low byte)
15
16     uint32_t periodicity;               // Period in ms (0 indicates aperiodic traffic)
17     uint32_t aperiodic_upper_bound;    // Upper bound for aperiodic traffic in ms
18     uint32_t aperiodic_lower_bound;    // Lower bound for aperiodic traffic in ms
19     uint32_t delta;                    // The delay bound delta in ms
20 } 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 UUIDs as the NRF52840DKs may be replaced over time

```
22 typedef struct
23 {
24     uint8_t node_id;           // ID of the current node
25     pattern_t p[TB_NUMPATTERN]; // Up to TB_NUMPATTERN parallel configurations
26 } 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

```

7  typedef struct
8  {
9      uint8_t traffic_pattern; // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10     uint8_t source_id[TB_NUMNODES]; // Only source_id[0] is used for p2p/p2mp
11     uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12     uint8_t msg_length; // Message length in bytes in/to EEPROM
13     uint8_t msg_offsetH; // Message offset in bytes in EEPROM (high byte)
14     uint8_t msg_offsetL; // Message offset in bytes in EEPROM (low byte)
15
16     uint32_t periodicity; // Period in ms (0 indicates aperiodic traffic)
17     uint32_t aperiodic_upper_bound; // Upper bound for aperiodic traffic in ms
18     uint32_t aperiodic_lower_bound; // Lower bound for aperiodic traffic in ms
19     uint32_t delta; // The delay bound delta in ms
20 } 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**

```

139 do{
140     m_xfer_done = false;
141     err_code = nrf_drv_twi_tx(&m_twi, EEPROM_ADDR, eeprom_by_address, 2, true);
142     if(NRF_SUCCESS != err_code)
143         continue;
144     while(!m_xfer_done);
145     m_xfer_done = false;
146     err_code = nrf_drv_twi_rx(&m_twi, EEPROM_ADDR, val, len);
147 }while(NRF_SUCCESS != err_code);
148 while(!m_xfer_done);

```

```

7 typedef struct
8 {
9     uint8_t traffic_pattern;           // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10    uint8_t source_id[TB_NUMNODES];    // Only source_id[0] is used for p2p/p2mp
11    uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12    uint8_t msg_length;                // Message length in bytes in/to EEPROM
13    uint8_t msg_offsetH;               // Message offset in bytes in EEPROM (high byte)
14    uint8_t msg_offsetL;               // Message offset in bytes in EEPROM (low byte)
15
16    uint32_t periodicity;               // Period in ms (0 indicates aperiodic traffic)
17    uint32_t aperiodic_upper_bound;    // Upper bound for aperiodic traffic in ms
18    uint32_t aperiodic_lower_bound;    // Lower bound for aperiodic traffic in ms
19    uint32_t delta;                   // The delay bound delta in ms
20 } pattern_t;

```

Input Parameters provided by the Testbed

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

```

139 do{
140     m_xfer_done = false;
141     err_code = nrf_drv_twi_tx(&m_twi, EEPROM_ADDR, eeprom_by_address, 2, true);
142     if(NRF_SUCCESS != err_code)
143         continue;
144     while(!m_xfer_done);
145     m_xfer_done = false;
146     err_code = nrf_drv_twi_rx(&m_twi, EEPROM_ADDR, val, len);
147 }while(NRF_SUCCESS != err_code);
148 while(!m_xfer_done);

```

```

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

```

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>

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

```

7 typedef struct
8 {
9     uint8_t traffic_pattern;
10    uint8_t source_id[TB_NUMNODES];
11    uint8_t destination_id[TB_NUMNODES];
12    uint8_t msg_length;
13    uint8_t msg_offsetH;
14    uint8_t msg_offsetL;
15
16    uint32_t periodicity;
17    uint32_t aperiodic_upper_bound;
18    uint32_t aperiodic_lower_bound;
19    uint32_t delta;
20 } pattern_t;

```

// 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
// Only source_id[0] is used for p2p/p2mp
// Only destination_id[0] is used for p2p/mp2p
// Message length in bytes in/to EEPROM
// Message offset in bytes in EEPROM (high byte)
// Message offset in bytes in EEPROM (low byte)
// Period in ms (0 indicates aperiodic traffic)
// Upper bound for aperiodic traffic in ms
// Lower bound for aperiodic traffic in ms
// The delay bound delta in ms

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

```
7 typedef struct
8 {
9     uint8_t traffic_pattern;           // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10    uint8_t source_id[TB_NUMNODES];    // Only source_id[0] is used for p2p/p2mp
11    uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12    uint8_t msg_length;                 // Message length in bytes in/to EEPROM
13    uint8_t msg_offsetH;                 // Message offset in bytes in EEPROM (high byte)
14    uint8_t msg_offsetL;                 // Message offset in bytes in EEPROM (low byte)
15
16    uint32_t periodicity;                // Period in ms (0 indicates aperiodic traffic)
17    uint32_t aperiodic_upper_bound;     // Upper bound for aperiodic traffic in ms
18    uint32_t aperiodic_lower_bound;     // Lower bound for aperiodic traffic in ms
19    uint32_t delta;                     // The delay bound delta in ms
20 } 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

```
7  typedef struct
8  {
9      uint8_t traffic_pattern;           // 0:unused, 1:p2p, 2:p2mp, 3:mp2p, 4: mp2mp
10     uint8_t source_id[TB_NUMNODES];    // Only source_id[0] is used for p2p/p2mp
11     uint8_t destination_id[TB_NUMNODES]; // Only destination_id[0] is used for p2p/mp2p
12     uint8_t msg_length;                 // Message length in bytes in/to EEPROM
13     uint8_t msg_offsetH;                 // Message offset in bytes in EEPROM (high byte)
14     uint8_t msg_offsetL;                 // Message offset in bytes in EEPROM (low byte)
15
16     uint32_t periodicity;                 // Period in ms (0 indicates aperiodic traffic)
17     uint32_t aperiodic_upper_bound;      // Upper bound for aperiodic traffic in ms
18     uint32_t aperiodic_lower_bound;      // Lower bound for aperiodic traffic in ms
19     uint32_t delta;                       // The delay bound delta in ms
20 } 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

```
22 typedef struct
23 {
24     uint8_t node_id;                // ID of the current node
25     pattern_t p[TB_NUMPATTERN];    // Up to TB_NUMPATTERN parallel configurations
26 } config_t;
27
```

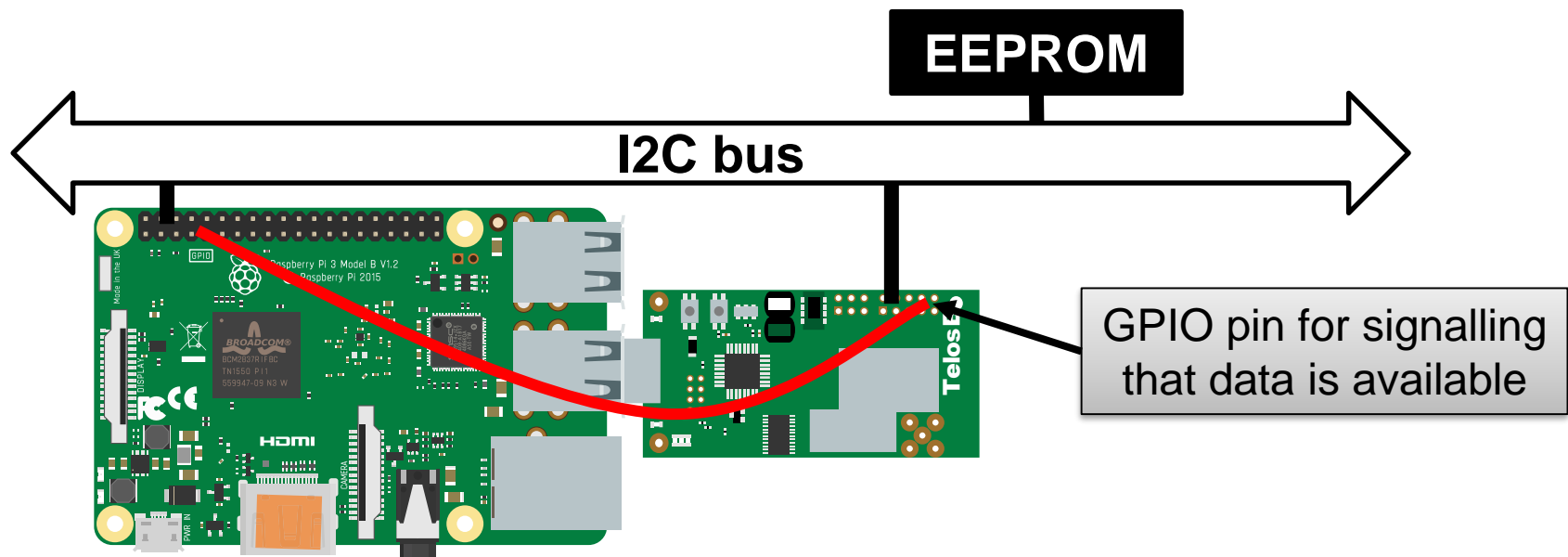
Printing Helper Function

- `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)

```
79 void
80 print_testbed_config(config_t* cfg)
81 {
82     printf("Testbed configuration:\n");
83     uint8_t i;
84     for (i=0; i<TR_NIMPATTERN; 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 0×50 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 on the Raspberry Pi
e.g. Pin P2.6 (GPIO3) 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

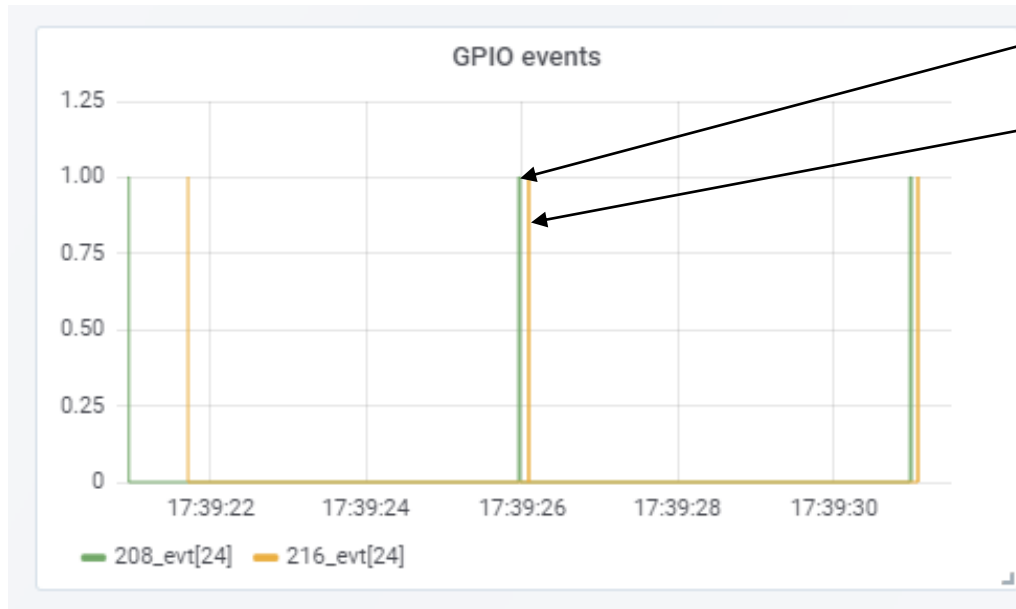
EEPROM Communication

- 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



Source can read new data from EEPROM

Destination has written data to EEPROM

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


[Show Layout](#)

~0.0min

[Create Job](#)

- 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

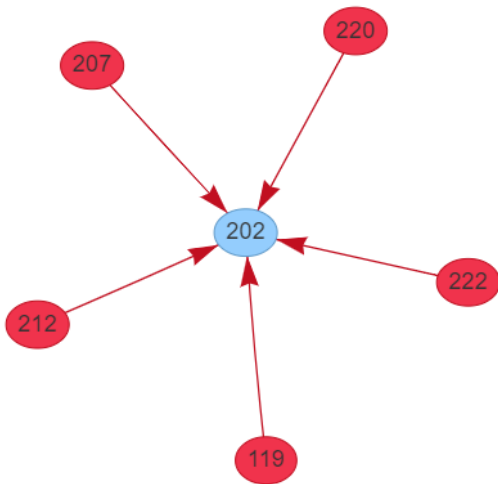
[Show Layout](#)

~0.0min

[Create Job](#)

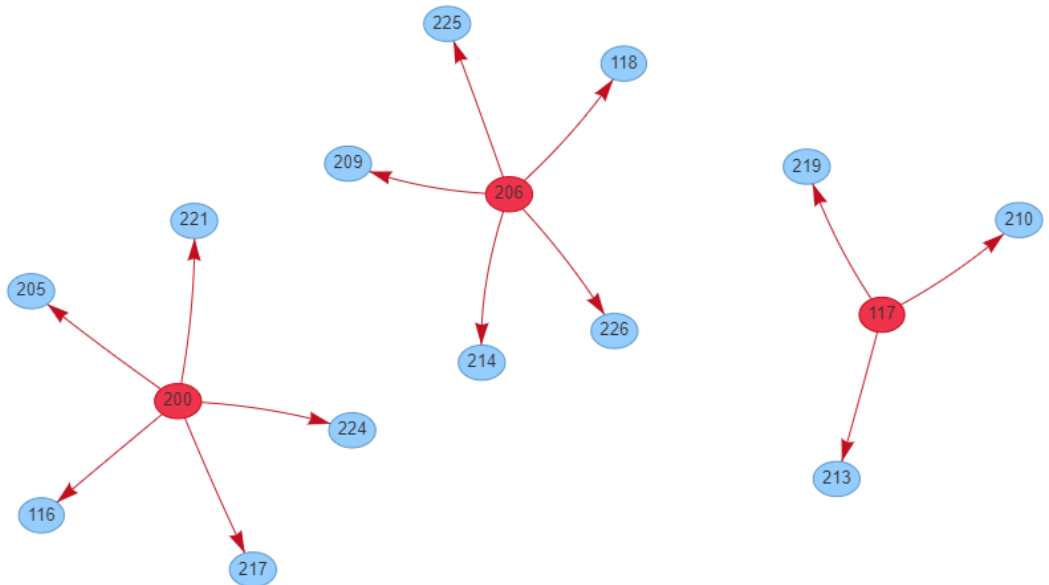
- How to interpret the available node layouts

Benchmark Suite Tmote Sky Data Collection v1
Node Layout Node Layout 1



- 207, 220, 212, 119, and 222 are source nodes
 - 202 is the destination node
- (These nodes are not necessarily in range)

Benchmark Suite Tmote Sky Dissemination v1
Node Layout Node Layout 1



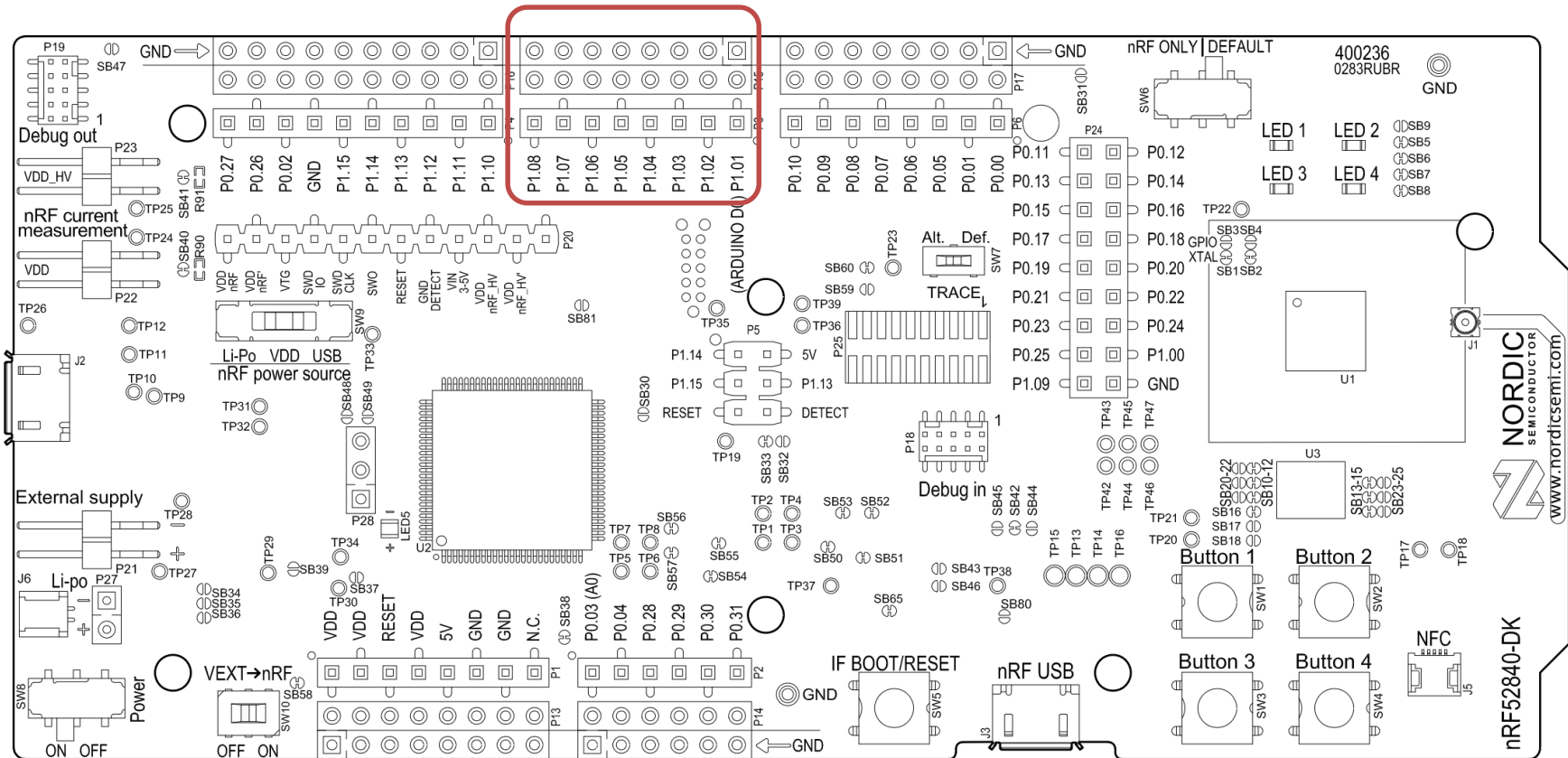
Three source nodes (200, 206, and 117)
transmitting to several destination nodes
(These nodes are not necessarily in range)

Hardware Details

(nRF52840DK)

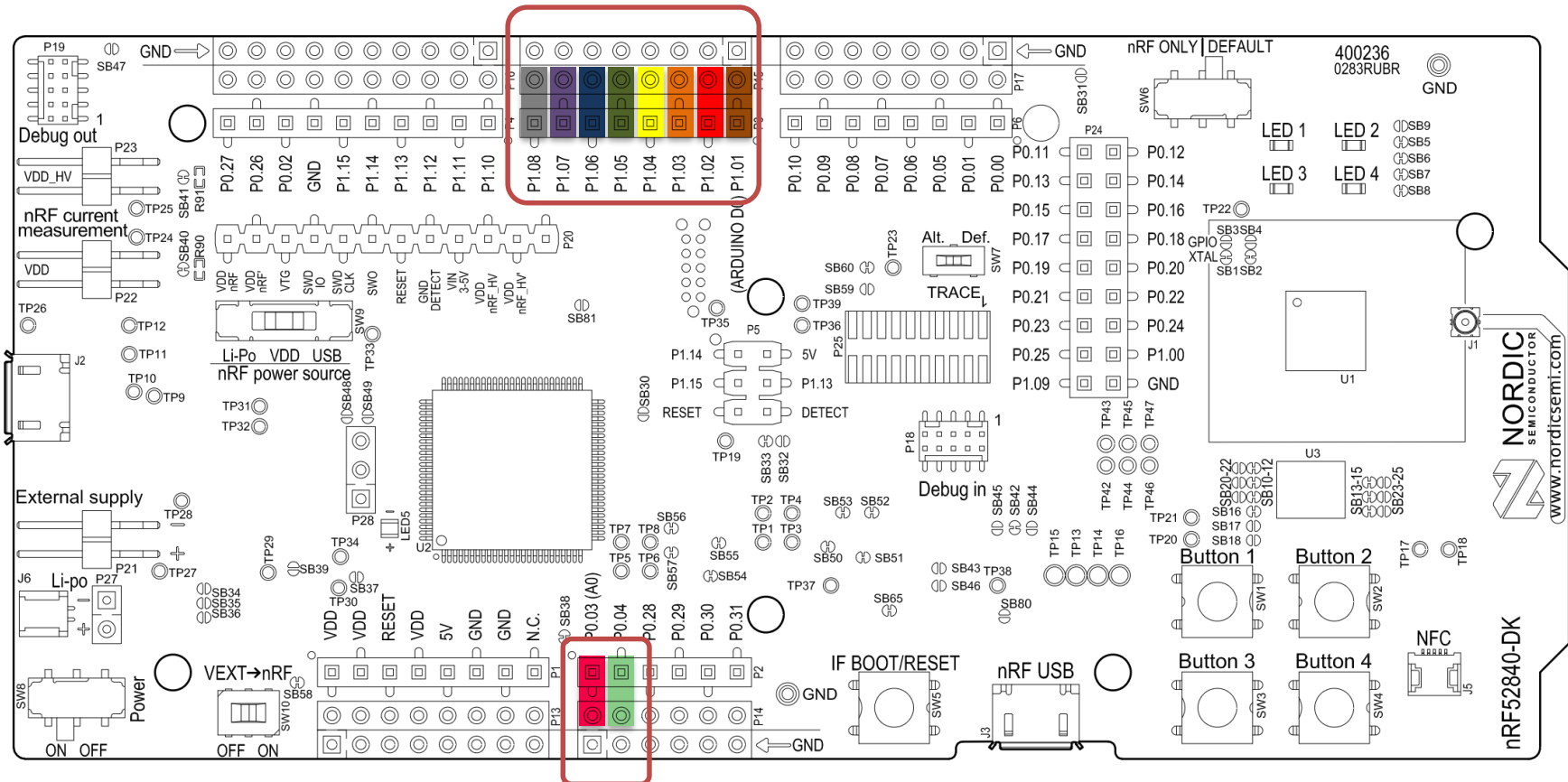
GPIO Pins

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

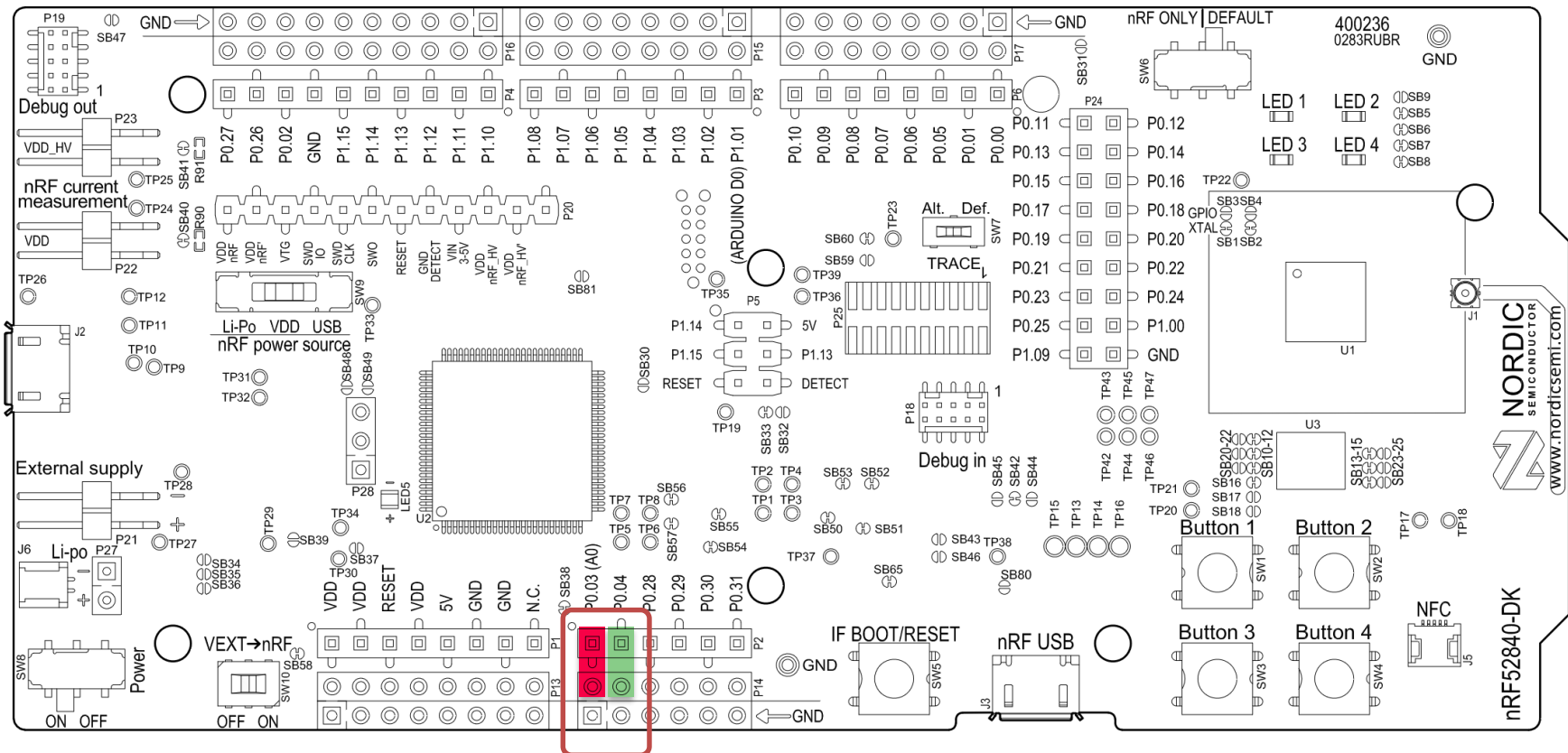


GPIO Pins

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

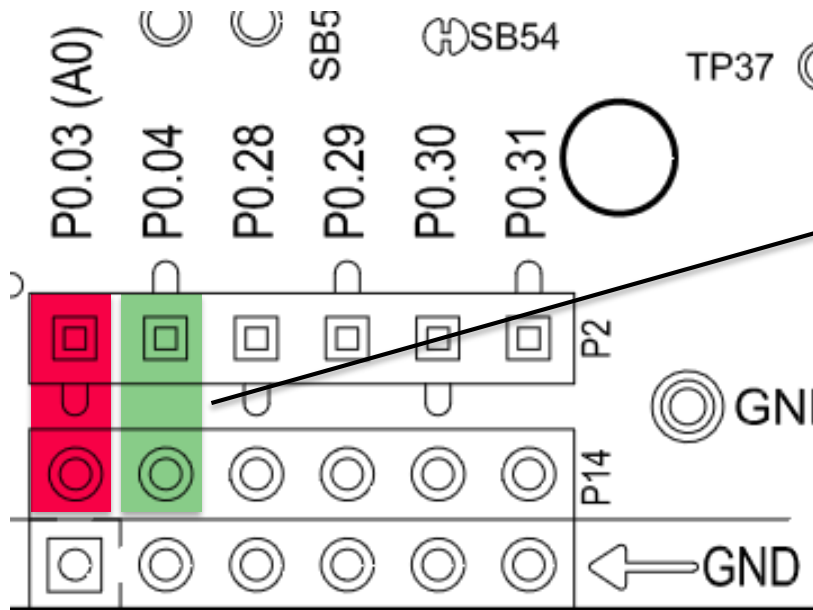


Sensor Node	Grafana
P1.01/D0	GPIO 17
P1.02/D1	GPIO 4
P1.03/D2	GPIO 18
P1.04/D3	GPIO 27
P1.05/D4	GPIO 22
P1.06/D5	GPIO 23
P1.07/D6	GPIO 24
P1.08/D7	GPIO 25



I2C Pins

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

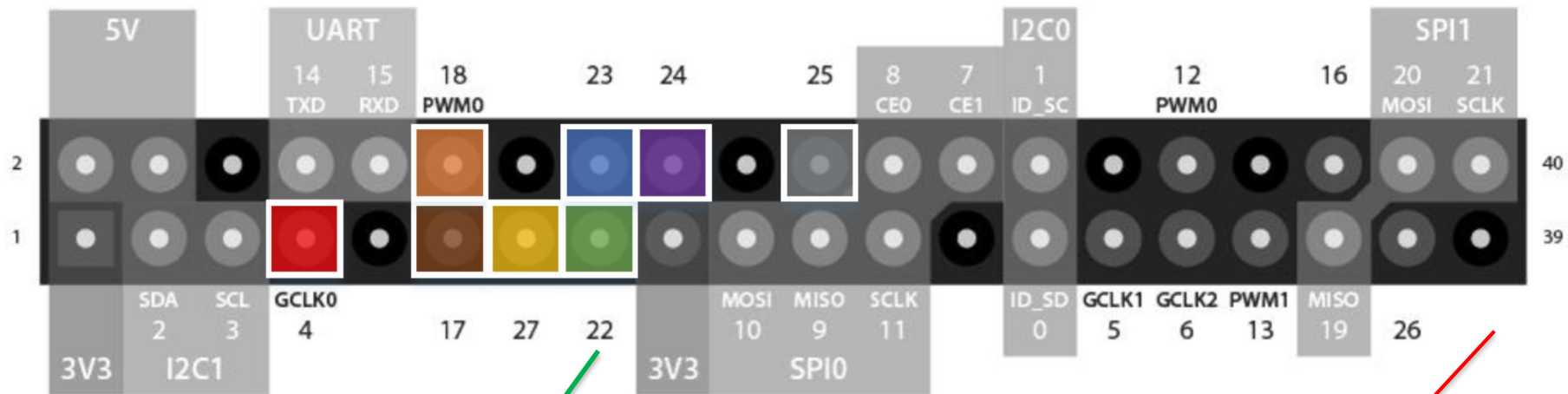


Sensor Node	I2C
P0.03/A0	SCL
P0.04/A1	SDA

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)

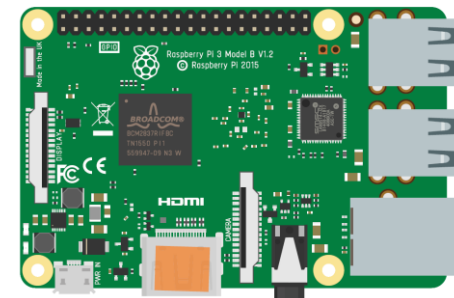
Raspberry Pi GPIO BCM numbering



(Pin 22 on
Raspberry Pi3)

P1.05/D4

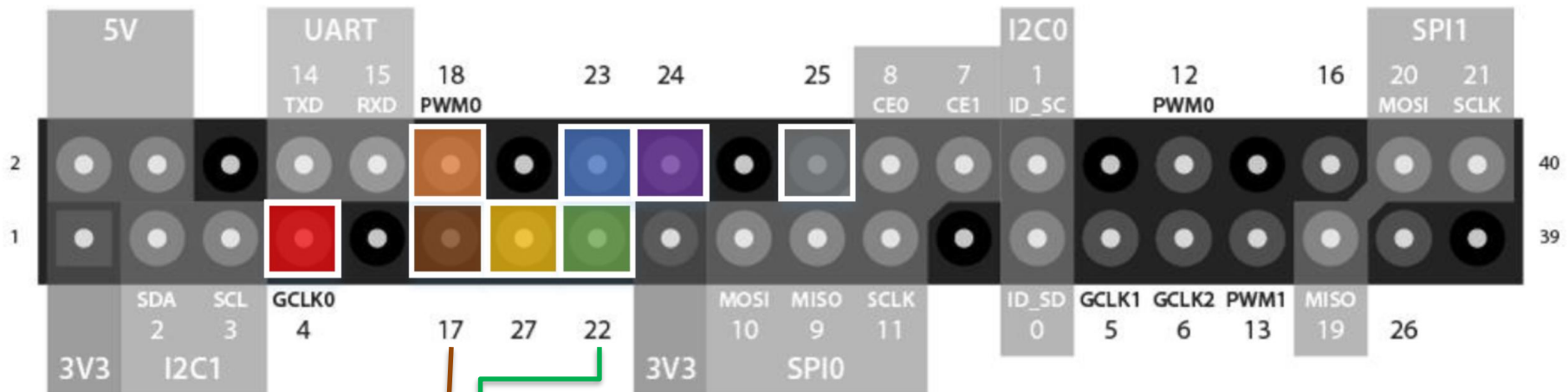
GPIO 22



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

GPIO Pins in Grafana

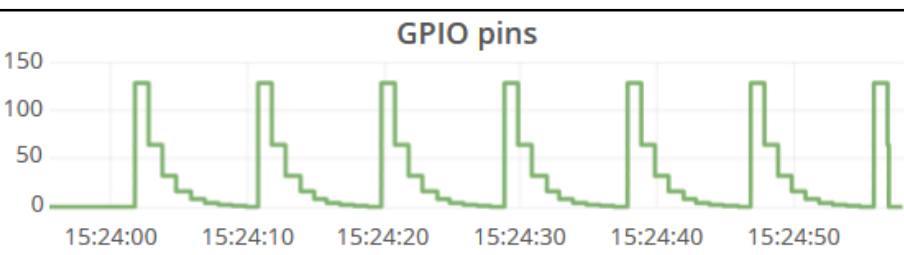
- In the “Overview of individual nodes” tab, the displayed “GPIO pins” numbers in Grafana are derived with the following mapping:

- 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

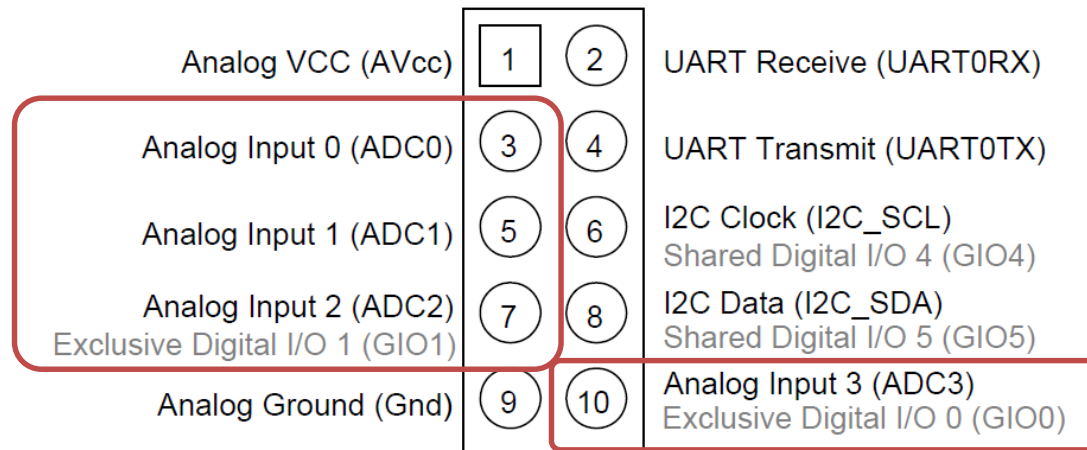


Hardware Details

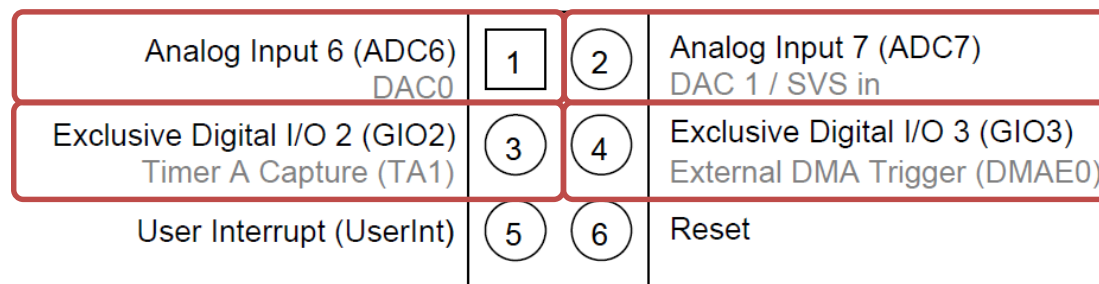
TelosB Sky

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

1	2
3	4
5	6
1. DAC0	6.6 (Grey)
2. SVSIn	6.7 (Green)
3. GIO2	2.3 (Blue)
4. GIO3/SVSout	2.6 (Purple)
5. UserINT	-
6. RESET	Reset (White)

1	2
3	4
5	6
7	8
9	10
1. DVCC	-
2. UART0RX	-
3. ADC0	6.0 (Brown)
4. UART0TX	-
5. ADC1	6.1 (Red)
6. I2C_SCL	-
7. ADC2/GIO1	6.2 (Orange)
8. I2C_SDA	-
9. GND	-
10. ADC3/GIO0	6.3 (Yellow)

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

```
//ADC1
P6SEL &= ~(BIT1);
P6DIR |= BIT1;
```

```
//ADC2
P6SEL &= ~(BIT2);
P6DIR |= BIT2;
```

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

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

```
//GIO2
P2SEL &= ~(BIT3);
P2DIR |= BIT3;
```

```
//GIO3
P2SEL &= ~(BIT6);
P2DIR |= BIT6;
```

```
//ADC6 / DAC0
P6SEL &= ~(BIT6);
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

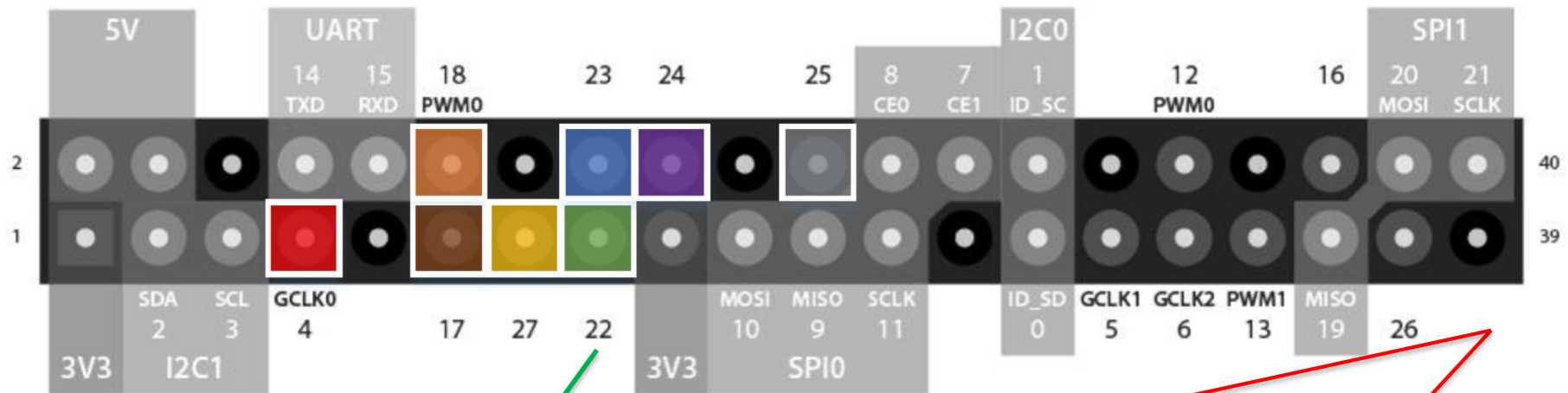
1	2	1. DAC0	6.6 (Grey)	-
2	3	2. SVSin	6.7 (Green)	-
3	4	3. GIO2	2.3 (Blue)	-
4	5	4. GIO3/SVSout	2.6 (Purple)	-
5	6	5. UserINT	-	-
6		6. RESET	Reset (White)	-

1	2	1. DVCC	-	-
2	3	2. UART0RX	-	-
3	4	3. ADC0	6.0 (Brown)	-
4	5	4. UART0TX	-	-
5	6	5. ADC1	6.1 (Red)	-
6	7	6. I2C_SCL	-	-
7	8	7. ADC2/GIO1	6.2 (Orange)	-
8	9	8. I2C_SDA	-	-
9	10	9. GND	-	-
10		10. ADC3/GIO0	6.3 (Yellow)	-

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)

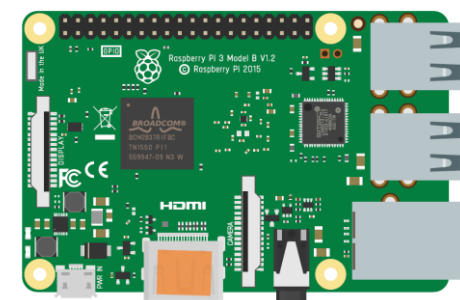
Raspberry Pi GPIO BCM numbering



(Pin 22 on
Raspberry Pi3)

ADC7/SVSin

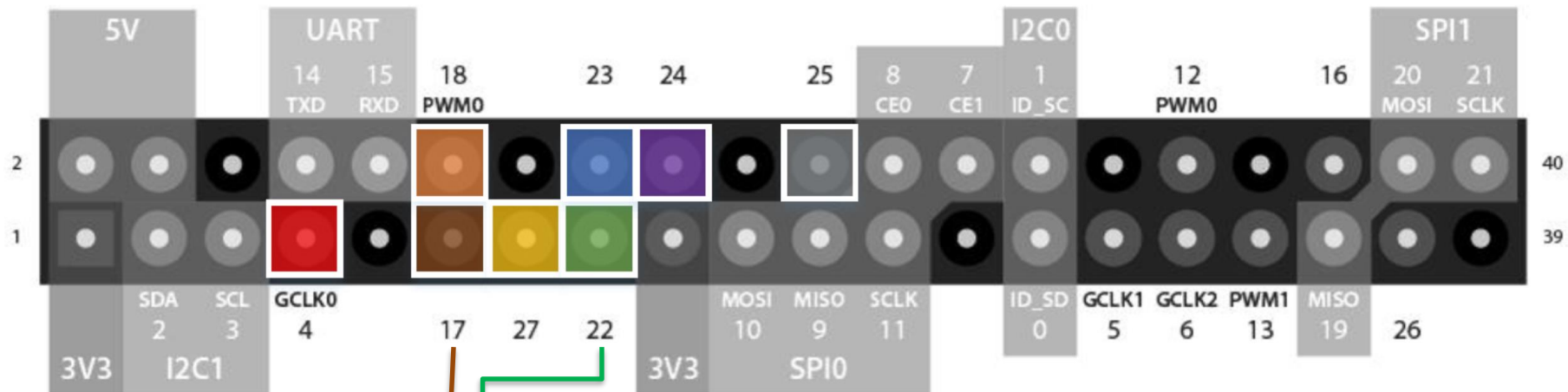
GPIO 22



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)

Raspberry Pi GPIO BCM numbering



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

GPIO Pins in Grafana

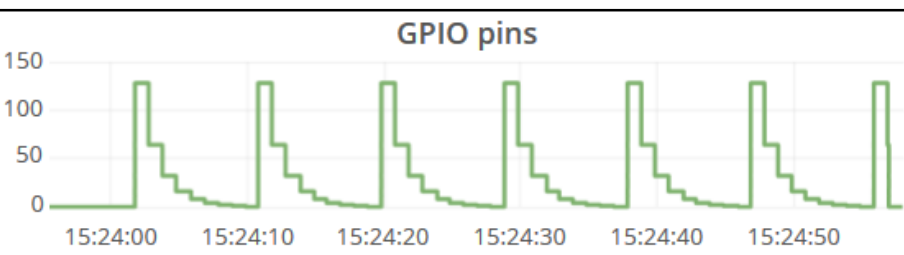
- In the “Overview of individual nodes” tab, the displayed “GPIO pins” numbers in Grafana are derived with the following mapping:

- 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