Communication protocol for the OpenShoe modules

John-Olof Nilsson

January 30, 2015

This document

This documents describes how to control and communicate with the OpenShoe modules. The general structure of the packages and the logics of the communication are described and details of the different commands, states, and processing functions and their functionality are given.

The document describes the communication protocol as of repository revisions 2f7ce4c and e48a22e for the runtime environment and the algorithm library, respectively. For earlier revisions there is no documentation apart from the code itself. However, the communication logic is similar but some commands differ.

If something doesn't work as described, most likely the documentation is wrong or the documentation describes the functionality of an older or newer version of the code. Have a look in the embedded code and update your module! If things still does not make sense, if relevant information is missing, or if you have any suggestions concerning this document, send an e-mail to openshoe@ee.kth.se.

Contents

1 Introduction

The OpenShoe modules (MIMU22BT, MIMU3333, MIMU4444) are controlled by a set of commands. The commands can be sent over either communication interfaces, USB or Bluetooth (UART). The commands supported by the embedded code and their functionality is described in this document.

A command can essentially do three different things: it can request a state a single time or at some rate, it can change a state of the module, and it can manipulate the process function sequence to modify the processing ran at each time instant. The behaviour of a command is controlled by a connected response function, which is run in response to the command being received by the module, and the command arguments which are passed to the response function.

It is possible for the process functions in the process sequence to modify the process sequence itself and to request the output of some state. For example, a process function may request a state to be output when some condition detected by the function appears. There are numerous utility functions and commands implemented which will perform such actions.

2 Setting up the communication

The USB is configured to appear as a virtual com-port on connection (USB CDC). For Windows an inf-file is required for setup (found here) the first time the device is connected. For Linux and Android the setup should be automatic.

For communication over Bluetooth the device need to be paired with the application platform. The device appears as a virtual com-port (SPP profile). Possibly it will also appear as a headset. This interface can be ignored. No special setup should be required. For Android, the module will be enumerated with a long id-number. This name can be found on Windows if you look at properties for your paired module.

3 Command and response structures

All the commands and responses start with a single byte header and end with a two byte checksum. The header is different for different commands and different responses. All data is in big-endian format. The structure of commands and responses and how to calculate the checksum is described in the following subsections. The commands are described in hexadecimal.

3.1 Command structure

The commands start with a one byte command specific header followed by a varying size payload and a two byte checksum. The payload can be different for different commands but is fixed for a specific command. The command structure is illustrated below. Each block correspond to one byte in the serial command.

CH	···	···	···	···	CK	CK

CH correspond to the single byte command header and the two CK to the two checksum bytes. The payload in between contains the arguments of the command and is indicated by ... bytes. A single argument may span multiple bytes.

3.2 Response structure

The response to a command is of two types. A command acknowledgement (ACK) or a data package. The acknowledgement to a command is a (data) package with a single byte header 0xA0, a single byte payload corresponding to the received command header and a two byte checksum. The acknowledgement to the command 03 (ping) will look like

A0	03	00	A3

The data packages are transmitted at some even rate, as a result of processing set up by some command or triggered by some configured condition. All data packages have a common structure. The responses start with a one byte header 0xAA followed by a two byte package number (N1 and N2), a single byte payload size in bytes (SZ), the payload, and a two byte checksum. The response structure is illustrated below.

AA	N1	N2	SZ	···	···	···	CK	CK

Note that the payload size is only a single byte and for large packages this byte may overflow. However, currently the only sensible case during which an overflow may occur is when raw IMU data from individual IMUs are requested. Since such data will only be requested for testing, such an overflow is acceptable and we have chosen to stay with a single byte payload size.

3.3 Checksum

The checksum is a simple 16-bit modular addition of all proceeding bytes in the command/response stored in big-endian format.

4 Reception and transmission logics

The communication logics implemented by the module is described in the following subsections.

4.1 Command reception

When receiving a byte, the module will check if this byte corresponds to a header. If not it will simply discard the byte and return to listen for a header. If it is a header it will check how many bytes to expect and try to read this number of bytes. If it cannot read the number of bytes within a timeout limit, it will discard the bytes it has read and return to listen for a new header. If the number of expected bytes are received, it will check the checksum. If the checksum is valid, it will try to parse the payload. If the payload is valid it will pass it to a response function connected to each command. If the checksum is valid, the parsing succeeds, and the command is not a package acknowledgement, and ACK will be prepared to be sent back to the application platform. Following this the module will return to listen for a new header.

Commands can be simultaneously received over both the USB and Bluetooth interfaces. The responses to the commands will be sent back over the same interface as they were received. The exception is the debug functions where the response interface may be chosen.

4.2 Response transmission

At each time instant the module will check if there is any ACK and states it should output to the user. If the latter is the case, it will assemble the data package in a buffer. An ACK is prepared by the command reception logics and reside in a separate buffer. In the data package, the states will be added in the order of their IDs. Once a package is assembled the package number and the size is added and the package number counter incremented. The buffer will then be pushed the ACK-buffer and the data package buffer to an output buffer. If the output buffer is full, the packages will be dropped.

For the wireless interface there is two modes of communication, a lossy mode and a lossless mode. In the lossy mode, which is identical to the USB, the data will simply be transmitted and forgotten about. In the lossless mode, the package will be added to a package que. The module will then keep transmitting the oldest package until it get a corresponding package acknowledgement back after which it will be removed from the que and the second oldest package transmitted. If the modules receives no package acknowledgements, the package que will fill up and packages will eventually be dropped.

5 Commands

On the following pages, the different commands are detailed. The commands comes in 4 groups: auxiliary, debugging, request data, request processing and combined commands. The auxiliary commands are composed of all "small" commands. The debugging commands are used to test and debug processing functions on the module. The request data commands will give some type of output of the state of the module. The processing request commands will make some processing being run on the module. This may also trigger some output. The combined commands will request some output and some processing. Typically, the combined commands are shortcuts to achieve behaviour which could have been attained by a combination of other commands.

The command header is given in the section title. The command argument and the command itself is described. Finally, examples of the command itself and responses are given for each command.

5.1 Package acknowledgement (0x01)

5.1.1 Arguments

1. Package number (2 bytes)

5.1.2 Description

Acknowledgement of reception of package sent back to module to signal that user has received the package indicated in the argument. This is necessary when using the lossless communication mode for the wireless interface, or the module will just keep sending the oldest package.

Being an acknowledgement in itself, this command will not be followed by an acknowledgement from the module. The action from the module is that the package (if the package indicated by the argument is the first package in the que) is removed from the package que.

5.1.3 Example command

01	00	01	00	02

The above command will signal the the user has received package with number 0x0001.

5.1.4 Example response

No response is given by the module.

5.2 Ping (0x03)

5.2.1 Arguments

None.

5.2.2 Description

Pings the module which will respond with an ACK. The module will take no further action.

5.2.3 Example command

03	00	03

This is the only form of the command.

5.2.4 Example response

a0	03	00	a3

No further response to the command will be given.

5.3 Module ID (0x04)

5.3.1 Arguments

None.

5.3.2 Description

The module will output a unique 15 byte serial number of the microcontroller. This may be used to identify the module. The response is subject to normal header, package numbers and checksums.

5.3.3 Example command

04	00	04

This is the only form of the command.

5.3.4 Example response

Acknowledgement

a0	04	00	a4

followed by

aa	00	01	0f	d1	f5	6f	00	51	4b	32	34	4e	20	20

20	ff	11	0c	05	bb

5.4 Setup debug processing and output (0x10)

5.4.1 Arguments

1. 8 single byte process function IDs (zero means no function)
2. 8 state IDs which will be output (zero gives no output)
3. 1 byte determining the interface the debug output should be transmitted over

5.4.2 Description

Configures and stores a process sequence of up to 8 process functions which will be run every time raw IMU data is pushed to the microcontroller (command 0x11). The process functions are run in the order they are given.

Configures a set of states which will be output every time raw IMU data is pushed to the microcontroller. If the first bit of the last argument is set, the output will be given over the USB. If the second bit of the last argument is set, the output will be given over Bluetooth. Both bits may be set.

5.4.3 Example command

The following command will setup process functions 0x10, 0x11, and 0x12 (the inertial frontend) to be run and the resulting combined inertial measurement with state id 0x13 to be output every time raw IMU data is pushed to the microcontroller.

10	10	11	12	00	00	00	00	00	13	00	00	00	00	00

00	00	00	ea

5.4.4 Example response

Acknowledgement

a0	10	00	b0

The data output is described in the command 0x20.

5.5 Input raw IMU data (0x11)

5.5.1 Arguments

1. 4 byte time stamp
2. 6 byte data for each IMU (the amount of data will vary depending on which board the code is compiled for)

5.5.2 Description

Input raw IMU data which is used to overwrite the data values read from the IMUs. Following this the response function will restore the process sequence set up by the setup-debug-processing command. This command is intended for testing (debugging) processing functions on the board.

If the setup command has not been run before this command or if the `store_and_empty_process_sequence()`-function is run in between, the restore will not work and the command may give unintended behaviour.

5.5.3 Example command

Example command for 4 IMUs

11	27	48	4d	94	00	62	00	8d	07	57	ff	e6	ff	f8

ff	d8	ff	6c	ff	92	f7	53	00	19	00	11	ff	fd	00

5e	00	83	07	9e	00	01	ff	e8	ff	b6	ff	7e	ff	85

f7	9c	ff	ff	ff	ef	ff	f1	1e	60

5.5.4 Example response

Acknowledgement

a0	10	00	b0

followed by output which is dependent on the setup command.

5.6 Set state (0x12-0x17)

5.6.1 Arguments

1. 1 byte state ID
2. 1-254 bytes with state values

5.6.2 Description

These commands will overwrite the value of the state, indicated by the first argument, with the first bytes of the second argument, corresponding to the size of the state. If the second argument is not sufficiently large, no overwrite will take place.

The different commands take a different size of the second argument:

1. 0x12 – 1 byte
2. 0x13 – 4 bytes
3. 0x14 – 12 bytes
4. 0x15 – 24 bytes
5. 0x16 – 48 bytes
6. 0x17 – 254 bytes

5.6.3 Example command

Setting the value of a single byte state (filter reset flag)

12	33	01	00	46

Setting the value of a 4-byte state (zero-velocity test statistics)

13	15	02	01	01	01	00	2d

Setting the value of a 12-byte state (position)

14	20	01	01	01	01	01	01	01	01	01	01	01	01	00
40														

5.6.4 Example response

Acknowledgement for set state of a single byte state

a0	0c	00	ac

No further response to the command will be given.

5.7 Request output of state (0x20)

5.7.1 Arguments

- 1 byte state ID
- 1 byte output mode

5.7.2 Description

Requests the state connected to the state ID to be output. The output mode byte controls how the state is output. The states are ordered according to their IDs in the response.

If the 6th bit of the output mode byte is set, the state will be output (pulled) a single time.

If the 6th bit of the mode selector is not set, the state will be output at a rate equal to the full rate (nominally 1000Hz) divided by 2^{x-1}, where x is the rate divider which is the 4 least significant bits of the mode selector. If the state is already set to be output at some rate, the command will change this rate. A rate divider of 0 will turn off the output.

If the request is sent over the wireless interface, the 5th bit of the output mode byte will control if lossy (not set) or lossless (set) transmission is used. This will apply to all subsequent output.

The output is automatically synchronized such that all output of lower rate coincide with those of higher rates. When a state is pulled, the state will be output immediately.

5.7.3 Example command

20	01	20	00	41

The IMU time stamp (ID 0x01) is polled a single time.

5.7.4 Example response

Acknowledgement

a0	20	00	c0

followed by

aa	06	76	04	1c	fb	65	d9	03	7f

5.8 Request output of multiple states (0x21)

5.8.1 Arguments

- 8 byte state IDs
- 1 byte output mode

5.8.2 Description

Requests the states connected to the state IDs to be output. The output mode byte controls how the state is output. The states are ordered according to their IDs in the response. This command is the same as 0x20 just with more IDs. If zeros are given instead of the IDs, no output will be given. Consequently, the command may be used to request output of up to 8 states.

For more information about the mode byte, see the 0x20 command.

5.8.3 Example command

21	10	11	15	16	00	00	00	00	04	00	71

The combined inertial readings 0x10 and 0x11 and test statistics 0x15 and 0x16 is requested with a rate of 125Hz (rate divider 0x04) (lossy if sent over Bluetooth).

5.8.4 Example response

Acknowledgement

a0	21	00	c1

followed by for example

aa	05	af	38	00	18	c4	00	00	0d	30	00	fc	2f	88

00	ff	fe	88	00	ff	fc	b8	00	ff	fd	94	00	00	1a

68	00	00	0b	80	00	fc	2e	38	00	ff	ff	80	00	ff

fa	b8	00	ff	fd	40	00	00	02	66	a4	00	00	01	73

17	84

5.9 Turn off all output (0x22)

5.9.1 Arguments

None.

5.9.2 Description

Turns off all even rate output and output setup to be triggered by a flag. The command also empties the package que of the Bluetooth interface. Output which is triggered by processing functions may still appear.

5.9.3 Example command

22	00	22

5.9.4 Example response

a0	22	00	c2

No further response to the command will be given.

5.10 Conditional output setup (0x23)

5.10.1 Arguments

1. 1 byte state ID of triggering state
2. 1 byte output mode
3. 8 bytes state IDs to be output

5.10.2 Description

Configures the state indicated by argument 1 (the first byte of the state being non-zero) to trigger the conditional output utility process function (0x03) to set an output with a mode given by argument 2 and states given by argument 3.

5.10.3 Example command

Set up the zero-velocity flag to trigger an output of itself.

23	17	20	17	00	00	00	00	00	00	00	00	71

5.10.4 Example response

a0	23	00	c3

No further response to the command will be given. This commands only setup the conditions under which the output is triggered and what the output is. To get the actual output, the process function 0x03 must be run and the condition most be true.

5.11 Output raw IMU data (0x28)

5.11.1 Arguments

1. 4 byte bit-field indicating from which out of 32 IMUs the data should be output
2. 1 byte output mode

5.11.2 Description

Tells the module to output the raw data read from the IMUs together with a common time stamp (4 bytes). The 32 bits in the bit-field correspond to the up to potentially 32 IMUs of the modules. A set bit will give the data from the corresponding IMU. If a bit is set which corresponds to a non-existing IMU, zeros will be output. (The states are still there but no data is ever written to the states.) The time stamp (state 0x01) is a 32-bit clock register read when the data is received from the IMUs. With a 64MHz clock frequency, this time stamp will wrap every 67s.

The output mode byte works the same way as for the general state output command but in addition to the rate divider (1st-4th bit), the lossy/lossless bit (5th bit) and the single/even rate transmission bit (6th bit), the 7th and 8th bit indicate if the raw inertial and/or the temperature readings of the IMUs should be output. The inertial data is 2×6 bytes per IMU and the temperature is 2 bytes per IMU.

5.11.3 Example command

28	00	00	00	0f	41	00	78

Request of the raw inertial data (not temperature) from the first 4 IMUs (the IMUs of the MIMU22BT modules) at a rate of 125Hz. If sent over Bluetooth, lossy transmission mode will be used.

5.11.4 Example response

Acknowledgement

a0	28	00	c8

Followed by for example

aa	19	d6	34	3d	78	02	5e	00	7f	00	02	f7	a6	00
01	ff	ea	00	09	00	09	ff	64	07	ac	ff	f5	ff	f0
ff	fb	ff	ef	ff	77	07	9d	00	11	00	01	00	26	00
95	ff	f8	f7	d2	ff	f5	ff	f2	00	19	1b	82		

5.12 Run processing function (0x30)

5.12.1 Arguments

1. 1 byte function ID
2. 1 byte array location

5.12.2 Description

Adds the process function indicated by the first argument to the array location indicated by the second argument. The array location must be between 0 and 10. A larger value will give no effect. A non-existing (or zero) function ID will remove any processing function in the array location.

5.12.3 Example command

30	10	00	36

Turns on the inertial frontend preprocessing part (locate the processing function in the first processing sequence slot) which will combine the raw inertial measurements from different IMUs to a single combined reading.

5.12.4 Example response

Acknowledgement

a0	30	00	d0

No further response will be given. However, some process function will trigger further output.

5.13 Run multiple processing functions (0x31)

5.13.1 Arguments

1. 8 byte function IDs

5.13.2 Description

Adds the 8 process functions indicated by the first argument to the array location 0-7. A non-existing (or zero) function ID will remove any processing function in the array location.

5.13.3 Example command

31	10	11	12	00	00	00	00	00	00	64

Turns on the inertial frontend.

5.13.4 Example response

Acknowledgement

a0	1f	00	bf

No further response will be given.

5.14 Stop all processing (0x32)

5.14.1 Arguments

None

5.14.2 Description

The command empties the processing sequence stopping all further processing.

5.14.3 Example command

32	00	32

This is the only form of the command.

5.14.4 Example response

Acknowledgement

a0	32	00	d2

No further response will be given.

5.15 Reset ZUPT-aided INS (0x33)

5.15.1 Arguments

None

5.15.2 Description

Starts (resets) the indefinite ZUPT-aided INS. A short initial alignment (256 sufficiently stationary samples) is performed after which the processing start. This can be combined with a request for the position state with some rate to let the module track the user without any processing on his side. However, in general, the step-wise version is preferable and this command is primarily kept for completeness.

5.15.3 Example command

33	00	33

This is the only form of the command.

5.15.4 Example response

Acknowledgement

a0	33	00	d3

No further response will be given.

5.16 Step-wise dead reckoning (0x34)

5.16.1 Arguments

None

5.16.2 Description

Starts (resets) step-wise dead reckoning which is just the ZUPT-aided INS but with resets and triggered output at every step. First a short initial alignment (256 sufficiently stationary samples) is performed after which the processing start. The initial alignment period will also let all buffers fill up and an initial bias estimate to be computed.

Once the processing is up and running, a displacement and heading change estimate together with corresponding error covariance estimate is pushed to the user every time the module becomes sufficiently stationary and at an even rate if stationary for an extended period. These ”steps“ can be used to perform dead reckoning on an application platform. For further details, see the article describing the OpenShoe modules.

5.16.3 Example command

34	00	34

This is the only form of the command.

5.16.4 Example response

Acknowledgement

a0	34	00	d4

Followed by step output like

aa	00	2a	3a	3c	ae	fe	a7	3e	7e	cb	be	bd	49	81

7d	be	96	59	a7	37	f0	24	e3	af	e0	31	de	31	1b

96	e7	32	f0	da	55	37	f0	19	49	32	da	48	e2	b1

19	bc	27	37	ef	b1	1b	ad	a1	52	4a	34	83	b8	df

00	0b	1e	c1

5.17 Start inertial frontend (0x35)

5.17.1 Arguments

None

5.17.2 Description

Places the inertial frontend processing function in location 0-2 in the processing sequence. These functions will combine the raw inertial readings from the different IMUs and compensate for the calibration (if available), calculate the test zero-velocity statistics and perform on-line bias calibration and conversion to floating point values with SI units. This command will enable the user to request combined inertial readings.

5.17.3 Example command

35	00	35

This is the only form of the command.

5.17.4 Example response

Acknowledgement

a0	35	00	d5

No further response will be given.

5.18 Restore process sequence setup (0x36)

5.18.1 Arguments

1. 1 byte state ID of triggering state

5.18.2 Description

Configures the state indicated by argument 1 (the first byte of the state being non-zero) to trigger the reset to a stored process sequence by the utility process function 0x04. Setting up and storing a process sequence can be done by adding the process functions to the process sequence and calling the store and empty process sequence command (0x37).

5.18.3 Example command

Set up the zero-velocity flag to trigger the restoration of the process sequence.

36	17	00	4d

5.18.4 Example response

a0	36	00	d6

No further response to the command will be given.

5.19 Store and empty process sequence (0x37)

5.19.1 Arguments

None.

5.19.2 Description

Saves the process sequence and empties the process sequence.

5.19.3 Example command

37	00	37

This is the only form of the command.

5.19.4 Example response

a0	37	00	d7

No further response to the command will be given.

5.20 Restore process sequence (0x38)

5.20.1 Arguments

None.

5.20.2 Description

Restore saved process sequence.

5.20.3 Example command

38	00	38

This is the only form of the command.

5.20.4 Example response

a0	38	00	d8

No further response to the command will be given.

5.21 Use as normal IMU (0x40)

5.21.1 Arguments

1. 1 byte output mode

5.21.2 Description

This command will configure the module to work as a normal IMU. The readings from the different IMUs will be combined and floating point readings (0x13) will be output according to the output mode byte.

For more information about the output mode byte, see command 0x20.

5.21.3 Example command

40	03	00	43

Request of combined inertial reading at full rate.

5.21.4 Example response

Acknowledgement

a0	40	00	e0

followed by output like

aa	00	01	1c	17	dd	3a	5d	3f	02	a2	4b	3c	cf	3c

7b	c1	15	8f	d2	bb	87	21	8c	bc	16	63	45	bb	ae

5c	d6	0d	7d

5.22 Use as normal IMU with online bias estimation (0x41)

5.22.1 Arguments

1. 1 byte output mode

5.22.2 Description

This command will configure the module to work as a normal IMU with online gyro bias estimation and compensation. Gyro biases are estimated based on time instants where the inertial readings in a time window shows a noise floor characteristics. The bias estimates will be subtracted from the combined inertial readings. Since the biases are estimated online there is a 127 sample delay (approximately 0.127s) in the received readings.

For more information about the output mode byte, see command 0x20.

5.22.3 Example command

41	03	00	44

Request of combined inertial reading at full rate.

5.22.4 Example response

Acknowledgement

a0	41	00	e1

The output will be the same as for 0x40 but the gyro bias should return to zero swiftly as the IMU becomes stationary.

6 States

The states are variables which can be accessed by the user. The states are also used by the processing functions to communicate so they represent, to a large extent, the information which is propagated through the processing. Of course, the complete state of the module is much larger but will be inaccessible without a debugger. Each state is identified by a single byte state ID. The states are declared extern in the header file belonging to the place where they are used. These header files are included in the file `system_state.c` which keeps a table of all states which is used by the system.

6.1 IMU time stamp (0x01)

State ID:	0x01
Type:	`uint32_t`
Size:	4 bytes

Time stamp taken before each read operation to the IMU.

6.2 Interrupt counter (0x02)

State ID:	0x02
Type:	`uint32_t`
Size:	4 bytes

A counter which is incremented by one each time the driving interrupt is set off.

6.3 Main loop time differential (0x03)

State ID:	0x03
Type:	`uint32_t`
Size:	4 bytes

Time differential indicating the execution time of the last main-loop.

6.4 Module ID (0x04)

State ID:	0x04
Type:	`char[15]` (see description below)
Size:	15 bytes

15 byte serial number of the module (of the microcontroller).

This state is not manifested by a proper variable but rather a pointer to an address in a write protected memory. Therefore the state cannot be set.

6.5 General purpose ID (0x05)

State ID:	0x05
Type:	`uint8_t`
Size:	1 bytes

General purpose ID state which can be set.

6.6 Combined inertial readings – preproc (0x10)

State ID:	0x10
Type:	`inert_int32` (struct)
Size:	24 bytes

Combined inertial readings provided directly by the inertial frontend preprocessing. The state contains 6 chunks of 4 bytes (`int32_t`). First 3 chunks of 4 bytes giving the combined specific force reading and then 3 chunks of 4 bytes giving the combined angular rate readings.

6.7 Combined inertial readings – statdet (0x11)

State ID:	0x11
Type:	`inert_int32` (struct)
Size:	24 bytes

Combined inertial readings provided by the inertial frontend statistics calculations. These readings are synchronized the provided test statistics and should be used in combination with these. The readings are used by the inertial frontend postporcessing function. The readings exhibits a delay of half the number of samples of the longest test statistics window. The state contains 6 chunks of 4 bytes (`int32_t`). First 3 chunks of 4 bytes giving the combined specific force reading and then 3 chunks of 4 bytes giving the combined angular rate readings.

6.8 Time stamp of 0x11 (0x12)

State ID:	0x12
Type:	`uint32_t`
Size:	4 bytes

Time stamp of the combined inertial readings 0x11. The source of these time stamps are 0x01. Compared with 0x01 this will show the delay of 0x12.

6.9 Combined inertial readings – floats (0x13)

State ID:	0x13
Type:	`inert_float` (struct)
Size:	24 bytes

Combined inertial readings converted to floating point. If only the frontend preprocessing is run, these readings will originate from 0x10. If the complete frontend is run, these readings will originate from 0x11 and exhibit the same delay. The state contains 6 chunks of 4 bytes (single precision floats). First floats giving the combined specific force reading and then 3 floats giving the combined angular rate readings.

6.10 Time differential (0x14)

State ID:	0x14
Type:	`float`
Size:	4 bytes

Time differential of the combined inertial readings 0x11. These time differentials are used in the mechanization.

6.11 Gaussian error model test statistics (0x15)

State ID:	0x15
Type:	`uint32_t`
Size:	4 bytes

Stationarity test statistics calculated based on an Gaussian error model.

6.12 Gaussian and bias error model test statistics (0x16)

State ID:	0x16
Type:	`uint32_t`
Size:	4 bytes

Stationarity test statistics calculated based on Gaussian plus bias error model.

6.13 Stationarity detection based on 0x15 (0x17)

State ID:	0x17
Type:	`bool`
Size:	1 bytes

Thresholded test statistics 0x15.

6.14 Stationarity detection based on 0x16 (0x18)

State ID:	0x18
Type:	`bool`
Size:	1 bytes

Thresholded test statistics 0x16.

6.15 Position (0x20)

State ID:	0x20
Type:	`float[3]`
Size:	12 bytes

Position state.

6.16 Velocity (0x21)

State ID:	0x21
Type:	`float[3]`
Size:	12 bytes

Velocity state.

6.17 Orientation (0x22)

State ID:	0x22
Type:	`float[4]`
Size:	16 bytes

Platform orientation in quaternion representation.

6.18 Filter error covariance (0x23)

State ID:	0x23
Type:	`float[45]`
Size:	180 bytes

Symmetric error covariance of the 9 states position, velocity, and orientation (euler angle representation). Since the matrix is symmetric only 45 values are stored.

6.19 Initialization-done flag (0x24)

State ID:	0x24
Type:	`bool`
Size:	1 bytes

Flag signaling when initialization (coarse initial alignment) is done.

6.20 Step (0x30)

State ID:	0x30
Type:	`float[4]`
Size:	16 bytes

Displacement and heading change extracted at filter reset.

6.21 Step error covariance (0x31)

State ID:	0x31
Type:	`float[10]`
Size:	40 bytes

Error covariance of 0x30.

6.22 Step counter (0x32)

State ID:	0x32
Type:	`uint16_t`
Size:	2 bytes

Counter counting the resets (steps) since power-up.

6.23 Filter-reset flag (0x33)

State ID:	0x25
Type:	`bool`
Size:	1 bytes

Flag signaling when the filtering has been reset (for step-wise dead reckoning).

6.24 Raw inertial readings from IMUs (0x40-0x5F)

State ID:	0x40-0x5F
Type:	`int16_t[6]`
Size:	12 bytes

Inertial readings, 3 specific force and 3 angular rate readings, from individual IMUs. Only the states corresponding to IMUs on the respective board will be filled up. The other states will remain zero. States of unmounted IMUs will be filled up with ones (due to pull-up resistors).

6.25 Raw temperature readings from IMUs (0x60-0x7F)

State ID:	0x60-0x7F
Type:	`int16_t`
Size:	2 bytes

Temperature readings from individual IMUs.

7 Processing functions

The processing functions are functions which the user can ask the module ro run in a specific order. The functions may employ other functions to complete its tasks and command response functions may also make the module run other functions. However, the process functions are the functions visible to the user. Each process function is identified by a single byte ID. The processing functions are briefly described below.

7.1 Store and empty process sequence (0x01)

Temporarily stores (for later restoration) and empties the process sequence.

7.2 Restore process sequence (0x02)

Restores the process sequence stored by the command 0x01. If no process sequence has been stored, the process sequence will be overwritten by an empty process sequence.

7.3 Empty process sequence (0x03)

Empties the process sequence, consequently also removing itself.

7.4 Conditional restore of process sequence (0x04)

Restores the process sequence given that a related state (flag) is true. For further information see command 0x36.

7.5 Conditional state output (0x05)

Flags one or more states for output given that a related state (flag) is true. For further information see command 0x23.

7.6 Single conditional state output (0x06)

Flags one or more states for output a single time given that a related state (flag) is true. For further information see command 0x23.

7.7 Conditional state output – counter (0x07)

Flags one or more states for output given that a related counter has surpassed a value. For further information see command 0x23.

7.8 Frontend preprocessing (0x10)

Preprocessing part to the inertial frontend. This function combines the raw inertial reading (0x40-0x5F) and performs the calibration compensation writing the results to 0x10. Finally, the combined inertial readings are converted to float values in 0x13.

7.9 Frontend statistics calculation (0x11)

Inertial data buffering and recursive zero-velocity statistics calculation. The functions has internal buffers in which the combined inertial reading 0x10 are buffered in order to calculate the statistics 0x15 and 0x16 and the related states 0x17 and 0x18. From the buffers, the function provides 0x11 and 0x12.

7.10 Frontend postprocessing (0x12)

Online bias estimation based on the calculated statistic 0x16, compensation applied to the values 0x11 and conversions of the results to floats 0x13. The function also calculates the time differential 0x14.

7.11 Mechanization (0x20)

Basic inertial mechanization.

7.12 Time update (0x21)

Time update of the Kalman filter.

7.13 Zero-velocity update (0x22)

Zero-velocity update of the Kalman filter.

7.14 Initial alignment (0x23)

Initial alignment for the aided inertial navigation. Rely on 0x17 and provided inertial reading in 0x13 to perform the initial alignment. Signal that the alignment is done by setting the flag 0x24.

7.15 Stepwise system reset (0x24)

Checks if the ZUPT-aided INS should be reset, saves the state, performs the reset and signal that a new step is available.