SmartAnthill Documentation

Release 0.0.0

Ivan Kravets

December 14, 2015

Contents

Release 0.0.0

Date December 14, 2015

Author Ivan Kravets

Home http://smartanthill.ikravets.com

Warning: The further work on the *SmartAnthill* Project has been moved to SmartAnthill 2.0.

SmartAnthill opens the door for people that are not familiar with electronics and micro-controller programming, but earlier had dream to use it. The main goal of *SmartAnthill* is to destroy the wall between usual user and hardware world. Thanks to this system we can combine the independent micro-devices or micro-based networks into general *SmartAnthill Network*.

You do not need to learn micro-programming languages, you do not need to install any IDE or Toolchain. All you need is to connect micro-device to *SmartAnthill*, to select capabilities that device should have and *"train it"* [1] to behave as the network device.

[1] The *"train it"* is that *SmartAnthill* creates unique *Embedded System* (firmware) for each supported micro-device and then installs it.

CHAPTER 1

Getting Started

1.1 Installation

1.1.1 Python & OS Support

SmartAnthill is written in Python and works with versions 2.6 and 2.7. *SmartAnthill* works on Unix/Linux, OS X, and Windows.

All commands below should be executed in Command-line application in your *OS*:

- *Unix/Linux/OS X* this is *Terminal* application.
- *Windows* this is Command Prompt (`cmd.exe`) application.

1.1.2 Super-Quick

To install or upgrade *SmartAnthill*, download get-smartanthill.py script.

Then run the following (which may require administrator access):

```
$ python get-smartanthill.py
```

An alternative short version for *Mac/Linux* users:

```
$ curl -L http://bit.ly/1qyr6K1 | python
```

On *Windows OS* it may look like:

```
C:\Python27\python.exe get-smartanthill.py
```

1.1.3 Full Guide

1. Check `python` version:

```
$ python --version
```

Note: *Windows OS Users* only:

1. Download Python and install it.
2. Download and install Python for Windows extensions.

3. Install *Python Package Index* utility using these instructions.
4. Add to *PATH* system variable `;C:\Python27;C:\Python27\Scripts;` and reopen *Command Prompt* (`cmd.exe`) application. Please read this article How to set the path and environment variables in Windows.

---

2. To install the latest release via PIP:

```
$ pip install smartanthill && pip install --egg scons
```

Note: If your computer does not recognize `pip` command, try to install it first using these instructions.

For upgrading the *SmartAnthill* to new version please use this command:

```
$ pip install -U smartanthill
```

1.2 Launching

SmartAnthill is based on Twisted and can be launched as *Foreground Process* as well as Background Process.

1.2.1 Foreground Process

The whole list of usage options for *SmartAnthill* is accessible via:

```
$ smartanthill --help
```

Quick launching (the current directory will be used as *Workspace Directory*):

```
$ smartanthill
```

Launching with specific *Workspace Directory*:

```
$  smartanthill --workspacedir=/path/to/workspace/directory
```

Check the *Configuration* page for detailed configuration options.

1.2.2 Background Process

The launching in the *Background Process* implements through `twistd` utility. The whole list of usage options for `twistd` is accessible via `twistd --help` command. The final *SmartAnthill* command looks like:

```
$ cd /path/to/workspace/directory
$ twistd smartanthill
```

1.3 Configuration

SmartAnthill uses JSON human-readable format for data serialization. This syntax is easy for using and reading.

The *SmartAnthill Configuration Parser* gathers data in the next order (steps):

1. Loads predefined *Base Configuration* options.
2. Loads options from *Workspace Directory*.

3. Loads *Console Options*.

Note: The *Configuration Parser* redefines options step by step (from #1 to #3). The *Console Options* step has the highest priority.

1.3.1 Base Configuration

The *Base Configuration* is predefined in *SmartAnthill System*. See config_base.json.

1.3.2 Workspace Directory

SmartAnthill uses `--workspacedir` for:

- finding user's specific start-up configuration options. They must be located in the `smartanthill.json` file. (Check the list of the available options here)
- finding the *Addons* for *SmartAnthill System*
- storing the settings about micro-devices
- storing the another working data.

For a start please **create empty directory** (like "project directory"). Later *SmartAnthill* will fill this folder with proper data.

Warning: The *Workspace Directory* must have Written Permission

1.3.3 Console Options

The simple options that are defined in *Base Configuration* can be redefined through console options for *SmartAnthill Application*.

The whole list of usage options for *SmartAnthill* are accessible via:

```
$ smartanthill --help
```

CHAPTER 2

Usage Documentation

CHAPTER 3

Developer Documentation

CHAPTER 4

Specification

4.1 Network

SmartAnthill Network is an independent micro-based and multi-master network that allows devices to communicate with each other. The micro-based device can be connected directly to *Network* through the different routers (for example, Serial Communication over Serial Port).

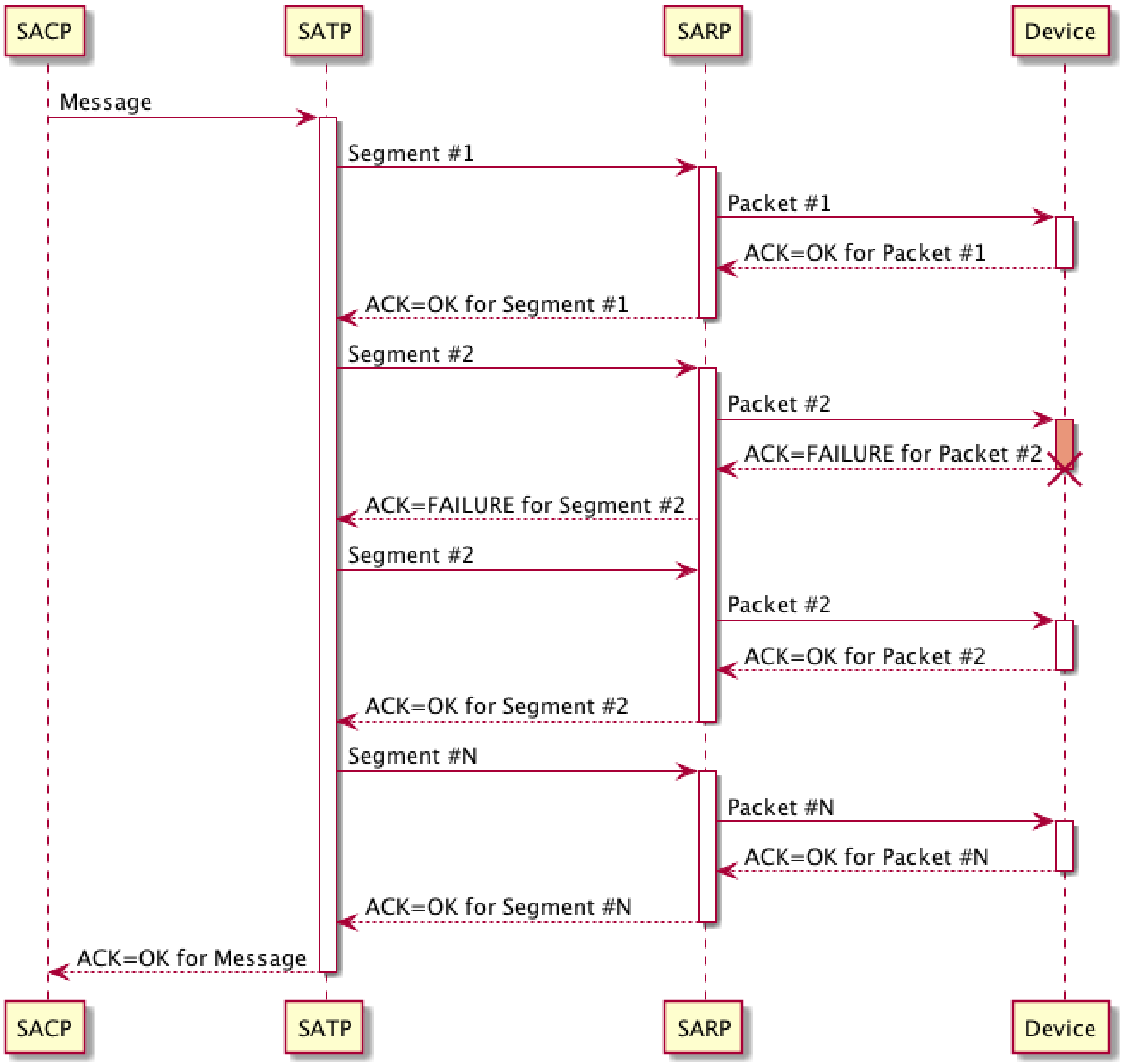
The key feature of the *Network* is communication with other networks. It can be extended with another *Network* or with Fieldbuses, like CAN.

4.1.1 Network Model

Comparasion with OSI Model

Layers	OSI-Model	SmartAnthill Model	Protocol	Data Unit	Service
7	Application				
6	Presentation	Application	*SACP*	Message	Queue
5	Session				
4	Transport	Transport	*SATP*	Segment	Queue
3	Network	Network	*SARP*	Packet	Router
2	Data-Link	Data-Link	CAN	Frame	Bridge
1	Physical	Physical	RS-232	Bit	

4.1.2 Protocols



Control Protocol (SACP)

Control Protocol (SACP) is a message based protocol with priority control. It resides at the *Application Layer* of the *Network Model*. The priority logic underlies the *Channel*. Each *Channel* has own *Data Classifier*.

Message structure

Part	Field name	Length (bits)	Description
Header	Channel	2	Channel ID (Priority)
	Data Classifier	6	Data Classifier ID
	SARP	16	SARP Header part
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	Length of Data in bytes
Payload	Data	0-14336	Maximum is 1792 bytes

`Channel (2 bits)` This is a *Channel ID* that specifies the priority of this *Message*. The smaller ID is, greater priority has the *Message*. For the whole channels list please check the *Channel Data Classifier*.

`Data Classifier (6 bits)` Check the *Channel Data Classifier*.

`SARP (16 bits)` This is an address information that contains *Source and Destination IDs* for *Routing Protocol (SARP)*.

`ACK (1 bit)` This is an *Acknowledgment* flag. If `ACK=1` then this *Message* should be confirmed by recipient about reception.

`TTL (4 bit)` Time to live (*TTL*) is a lifetime in seconds of *Message* in *Network*. The maximum value is 15 seconds. When *TTL* is up the `MessageLostException` will be raised.

`Data length (11 bits)` This is a length of *Payload* part in bytes. The *Message* can be empty (without *Payload*). In this situation when `Data length=0x0` *Payload* part is not presented in the *Message*.

`Data (0-14336 bits)` The maximum size of *Payload* part is 1792 bytes.

Note: This limitation was caused by maximum numbers of *Segments* from *Transport Protocol (SATP)*. `256 segments * 7 bytes of user data = 1792 bytes`

Transport Protocol (SATP)

Transport Protocol (SATP) resides between *Control Protocol (SACP)* and *Routing Protocol (SARP)* and operates with the two data units (*Message* and *Segment*). Therefore, he has bi-directional work.

Between *Application Layer* and *Transport Layer* of the *Network Model*, it divides into *Segments* the outbound *Message*. While between *Transport Layer* and *Network Layer* it assembles multiple inbound *Segments* into single *Message*.

Transport Protocol (SATP) is a reliable protocol. It can guarantee delivery of each *Segment* if source device asked for it. Also it can guarantee the integrity of final *Message* because *Transport Protocol (SATP)* knows about the order of each *Segment*.

Segment structure

Part	Field name	Length (bits)	Description
Header	SACP	8	SACP Header part
	SARP	16	SARP Header part
	SEG	1	Segmentation flag
	FIN	1	Final segment flag
	ACK	1	Acknowledgment flag
	Reserved	1	Must be set to 0x0
	Data length	4	Length of Data in bytes
Payload	Data	0-64	Maximum is 8 bytes
	CRC	16	Checksum

`SACP (8 bits)` These are the *Channel* and *Data Classifier* for *Control Protocol (SACP)*.

`SARP (16 bits)` This is an address information that contains *Source and Destination IDs* for *Routing Protocol (SARP)*.

`SEG (1 bit)` This is a *Segmentation* flag. If the *Message* is not segmented then `SEG=0` otherwise `SEG=1`.

Note: The service information about *Segments Order* is located in the first byte of *Data* field. Therefore it is followed that the maximum number of *Segments* is 256. The first *Segment* marks as `0x0`, the second as `0x1` and the last as `0xFF`

`FIN (1 bit)` It indicates that this *Segment* is final.

`ACK (1 bit)` This is an *Acknowledgment* flag. If `ACK=1` then this *Segment* should be confirmed by recipient about reception.

`Data length (4 bits)` This is a length of *Payload* part in bytes. The *Segment* can be empty (without *Payload*). In this situation when `Data length=0x0`, `SEG=0` and `FIN=1` *Payload* part is not presented in the *Segment*. The maximum size of *Payload* part is 8 bytes.

`Data (0-64 bits)` This is a *Payload* data. If `SEG=1` the first byte of the data will be used for *Segments Order* information and another 7 are available for user.

`CRC (16 bits)` The 16-bit checksum is used for error-checking of the *Header* and *Payload* parts.

Routing Protocol (SARP)

The main goal of the *Routing Protocol (SARP)* is to find a route and transfer a packet to destination device that located in the *Network*. The *Routing Protocol (SARP)* does not guarantee delivery. The only thing that it guarantees is integrity of the *Header* and the *Payload* data in the packet (based on CRC).

Packet structure

Part	Field name	Length (bits)	Description
	SOP	8	Start of packet
Header	SACP	8	SACP Header part
	Source	8	The source device ID
	Destination	8	The destination device ID
	SATP	3	SATP Header part
	Reserved	1	Must be set to 0x0
	Data length	4	The length of data in bytes
Payload	Data	0-64	Max 8 bytes
	CRC	16	Checksum
	EOF	8	End of packet

`SOP (8 bits)` It specifies the start of the packet. These 8 bits are equal to ASCII Start Of Heading (SOH) character `0x1`.

`SACP (8 bits)` These are the *Channel* and *Data Classifier* for *Control Protocol (SACP)*.

`Source (8 bits)` This is an *Identifier (ID)* of the source device. *Network* supports up to 255 devices. Each device has unique identifier from range 0-255. The device with `ID=0x0` corresponds to *Zero Virtual Device*.

`Destination (8 bits)` This is an *Identifier (ID)* of destination device. *Network* supports up to 255 devices. Each device has unique identifier from range 0-255. The device with `ID=0x0` corresponds to *Zero Virtual Device*.

`SATP (3 bits)` These are the *Segmentation*, *Final* and *Acknowledgment* flags for *Transport Protocol (SATP)*.

`Data length (4 bits)` This is a length of *Payload* data in bytes. The *Packet* can be empty (without *Payload*). In this situation `Data length=0x0` and *Payload* part is not present in the *Packet*. The maximum size of *Payload* part are 8 bytes.

`Data (0-64 bits)` This is a *Payload* part for *Transport Protocol (SATP)*.

`CRC (16 bits)` The 16-bit checksum is used for error-checking of the *Header* and *Payload* parts.

`EOF (8 bits)` It specifies the end of the packet. These 8 bits are equal to ASCII End of Transmission (SOH) character `0x17`.

4.1.3 Channel Data Classifier

<table>
<tr><th colspan="2">Channel (2 bits)</th><th colspan="2">Data Classifier (6 bits)</th></tr>
<tr><th>ID</th><th>Name</th><th>ID</th><th>Name</th></tr>
<tr><td rowspan="2">0x0</td><td rowspan="2">Urgent</td><td>0x00</td><td>Ping</td></tr>
<tr><td>0x0A</td><td>SegmentAcknowledgment</td></tr>
<tr><td>0x1</td><td>Event-Driven</td><td></td><td></td></tr>
<tr><td rowspan="6">0x2</td><td rowspan="6">Bi-Directional Communication (Request)</td><td>0x09</td><td>ListOperationalStates</td></tr>
<tr><td>0x0A</td><td>ConfigurePinMode</td></tr>
<tr><td>0x0B</td><td>ReadDigitalPin</td></tr>
<tr><td>0x0C</td><td>WriteDigitalPin</td></tr>
<tr><td>0x0D</td><td>ConfigureAnalogReference</td></tr>
<tr><td>0x0E</td><td>ReadAnalogPin</td></tr>
<tr><td rowspan="6">0x3</td><td rowspan="6">Bi-Directional Communication (Response)</td><td>0x09</td><td>ListOperationalStates</td></tr>
<tr><td>0x0A</td><td>ConfigurePinMode</td></tr>
<tr><td>0x0B</td><td>ReadDigitalPin</td></tr>
<tr><td>0x0C</td><td>WriteDigitalPin</td></tr>
<tr><td>0x0D</td><td>ConfigureAnalogReference</td></tr>
<tr><td>0x0E</td><td>ReadAnalogPin</td></tr>
</table>

Urgent

The channel with the highest priority. It uses for the critical tasks or operations.

Ping

Uses to test the reachability of *Network Device*. If device is reachable you will receive *SegmentAcknowledgment Segment*.

The *Message* by *Control Protocol (SACP)* should have the next structure:

<table>
<tr><th>Part</th><th>Field name</th><th>Length (bits)</th><th>Value</th></tr>
<tr><td rowspan="6">Header</td><td>Channel</td><td>2</td><td>0x00</td></tr>
<tr><td>Data Classifier</td><td>6</td><td>0x00</td></tr>
<tr><td>SARP</td><td>16</td><td>Routing Protocol (SARP) address information</td></tr>
<tr><td>ACK</td><td>1</td><td>Should be 0x01</td></tr>
<tr><td>TTL</td><td>4</td><td>Time to live</td></tr>
<tr><td>Data length</td><td>11</td><td>0x0</td></tr>
<tr><td>Payload</td><td>Data</td><td>0</td><td>Without Payload part</td></tr>
</table>

SegmentAcknowledgment

Uses for acknowledge that *Segment* from sender was received and verified.

The *Segment* by *Transport Protocol (SATP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x00
	Data Classifier	6	0x0A
	SARP	16	*Routing Protocol (SARP)* address information
	SEG	1	0x0
	FIN	1	0x1
	ACK	1	0x0
	Reserved	1	Must be set to 0x0
	Data length	4	0x2
Payload	Data	16	The *CRC* field from received *Packet*

Event-Driven

Bi-Directional Communication (Request)

ListOperationalStates

Retrieve a list with activated *Operational States* for specified device. For the result please read *ListOperationalStates* from *Bi-Directional Communication (Response)* channel.

The *Message* by *Control Protocol (SACP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x02
	Data Classifier	6	0x09
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x0
Payload	Data	0	Without *Payload* part

ConfigurePinMode

Configure the specified pin to behave either as an:

- `INPUT`
- `OUTPUT`
- `INPUT_PULLUP`
- `INPUT_PULLDOWN`

For the result please read *ConfigurePinMode* from *Bi-Directional Communication (Response)* channel.

The *Message* by *Control Protocol (SACP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x02
	Data Classifier	6	0x0A
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x2
Payload	Data	8	The number of the pin
		8	The mode of the pin (see table above)

Note: You can configure more than one Pin using single *Message*. Please use the next sequence of bytes in *Payload* part of *Message* -> `pin1, mode1, pin2, mode2, ..., pinN, modeN`

ReadDigitalPin

Read the value from a specified digital pin. For the result please read *ReadDigitalPin* from *Bi-Directional Communication (Response)* channel.

The *Message* by *Control Protocol (SACP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x02
	Data Classifier	6	0x0B
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The number of the pin

Note: You can read more than one Pin using single *Message*. Please use the next sequence of bytes in *Payload* part of *Message* -> `pin1, pin2, ..., pinN`

WriteDigitalPin

Write a `LOW` or a `HIGH` level to a digital pin. For the result please read *WriteDigitalPin* from *Bi-Directional Communication (Response)* channel.

The *Message* by *Control Protocol (SACP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x02
	Data Classifier	6	0x0C
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x2
Payload	Data	8	The number of the pin
		8	The level (`0x1=HIGH` or `0x0=LOW`)

Note: You can write to more than one Pin using single *Message*. Please use the next sequence of bytes in *Payload* part of *Message* -> `pin1, value1, pin2, value2, ..., pinN, valueN`

ConfigureAnalogReference

Configure the reference voltage used for analog input. The modes are:

- `DEFAULT`
- `INTERNAL`
- `INTERNAL1V1`
- `INTERNAL2V56`
- `INTERNAL1V5`

- `INTERNAL2V5`
- `EXTERNAL`

For the result please read *ConfigureAnalogReference* from *Bi-Directional Communication (Response)* channel.

The *Message* by *Control Protocol (SACP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x02
	Data Classifier	6	0x0D
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The mode (see table above)

ReadAnalogPin

Read the value from a specified analog pin. For the result please read *ReadAnalogPin* from *Bi-Directional Communication (Response)* channel.

The *Message* by *Control Protocol (SACP)* should have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x02
	Data Classifier	6	0x0E
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The number of the pin

Note: You can read more than one Pin using single *Message*. Please use the next sequence of bytes in *Payload* part of *Message* -> `pin1, pin2, ..., pinN`

Bi-Directional Communication (Response)

ListOperationalStates

The result of the request from *Bi-Directional Communication (Request)* channel and *ListOperationalStates*. The *Payload* part will contain the list of activated *Operational States*. Where each byte will be equal to *Channel Data Classifier ID*.

The *Message* by *Control Protocol (SACP)* will have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x03
	Data Classifier	6	0x09
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The *Channel Data Classifier ID*

Note: If device has more than one activated *Operational State* then the *Payload* part of *Message* will have the next sequence of bytes -> `cdcID1, cdcID2, ..., cdcIDN`

ConfigurePinMode

The result of the request from *Bi-Directional Communication (Request)* channel and *ConfigurePinMode*. The *Payload* part will contain the list of pins that was successfully configured with specified mode.

The *Message* by *Control Protocol (SACP)* will have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x03
	Data Classifier	6	0x0A
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The number of the pin

Note: If you specified more than one Pin using single *Message* then the *Payload* part of *Message* will have the next sequence of bytes -> `pin1, pin2, ..., pinN`

ReadDigitalPin

The result of the request from *Bi-Directional Communication (Request)* channel and *ReadDigitalPin*. The *Payload* part will contain the result from requested pins. The result value can be as `0x1` (high level) or `0x0` (low level).

The *Message* by *Control Protocol (SACP)* will have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x03
	Data Classifier	6	0x0B
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The value (`0x1` or `0x0`)

Note: If you specified more than one Pin using single *Message* then the *Payload* part of *Message* will have the next sequence of bytes -> `value1, value2, ..., valueN`

WriteDigitalPin

The result of the request from *Bi-Directional Communication (Request)* channel and *WriteDigitalPin*. The *Payload* part will contain the list of pins that was successfully updated with specified levels.

The *Message* by *Control Protocol (SACP)* will have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x03
	Data Classifier	6	0x0C
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The number of the pin

Note: If you specified more than one Pin using single *Message* then the *Payload* part of *Message* will have the next sequence of bytes -> `pin1, pin2, ..., pinN`

ConfigureAnalogReference

The result of the request from *Bi-Directional Communication (Request)* channel and *ConfigureAnalogReference*. The first byte of *Payload* part will contain `0x01` if the reference voltage was successfully configured, otherwise `0x00`.

The *Message* by *Control Protocol (SACP)* will have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x03
	Data Classifier	6	0x0A
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x1
Payload	Data	8	The result: `0x00` or `0x01`

ReadAnalogPin

The result of the request from *Bi-Directional Communication (Request)* channel and *ReadAnalogPin*. The *Payload* part will contain the result from requested pins. The result value can be between 0-1023 (for 10-bit ADC) or between 0-4095 (for 12-bit ADC).

The *Message* by *Control Protocol (SACP)* will have the next structure:

Part	Field name	Length (bits)	Value
Header	Channel	2	0x03
	Data Classifier	6	0x0E
	SARP	16	*Routing Protocol (SARP)* address information
	ACK	1	Acknowledgment flag
	TTL	4	Time to live
	Data length	11	0x2
Payload	Data	8	The MSB of result
		8	The LSB of result

Note: If you specified more than one Pin using single *Message* then the *Payload* part of *Message* will have the next sequence of bytes -> `MSB_value1, LSB_value1, MSB_value2, LSB_value2, ..., MSB_valueN, LSB_valueN`

4.1.4 Integration with CAN

CAN bus is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as aerospace, maritime, industrial automation and medical equipment (got from wiki).

Protocol

Network can be easy integrated with `CAN` because the protocols of these networks are frame-based. `CAN` resides on the *Data-Link Layer* of the *Network* Model and represented with data unit as *Frame*. While the *Network Layer* operates through *Routing Protocol (SARP)* and *Packet*. Therefore, *SARP* will work over `CAN Protocol 2.0B` (specification with extended message formats).

The *Data Length* field of the *Packet* from *SARP* is equivalent with `CAN` *Frame*. The *SARP* *Header* part can be converted to `CAN` *Extended Identifier (29 bit)*.

Frame structure

<table>
<tr><th>Part</th><th>Field name</th><th colspan="2">Length (bits)</th><th>Description</th></tr>
<tr><td rowspan="4">Header</td><td>SACP</td><td rowspan="4">29</td><td>8</td><td>SACP Data Classifier</td></tr>
<tr><td>SARP</td><td>16</td><td>SARP address information</td></tr>
<tr><td>SATP</td><td>3</td><td>SATP flags</td></tr>
<tr><td>Reserved</td><td>2</td><td>Must be set to 0x0</td></tr>
<tr><td>Length</td><td>Data length</td><td colspan="2">4</td><td>The length of data in bytes</td></tr>
<tr><td>Payload</td><td>Data</td><td colspan="2">0-64</td><td>Max 8 bytes</td></tr>
</table>

Note: The fields *Start of Frame*, *Cyclic redundancy check* and *End of Frame* are not presented in this structure because the CAN protocol has own implementation for its.

`SACP (8 bits)` The *Channel* and *Data Classifier* for *Control Protocol (SACP)*.

`SARP (16 bits)` The address information that contains *Source and Destination IDs* for *Routing Protocol (SARP)*.

`SATP (3 bits)` The *Segmentation*, *Final* and *Acknowledgment* flags for *Transport Protocol (SATP)*

`Data length (4 bits)` The length of *Payload* part in bytes. The *Frame* can be empty (without *Payload*). In this situation `Data length=0x0` and *Payload* is not presented in the *Frame*. The maximum size of *Payload* part is 8 bytes.

`Data (0-64 bits)` The *Payload* data for *Transport Protocol (SATP)*.

4.1.5 Zero Virtual Device

This is a virtual device in *Network* with `ID=0x0`.

4.2 System

4.2.1 Applications

- Connectivity

- Sensor aggregation
- Security and access control
- Home and building automation
- Industrial automation
- Human machine interface
- Lighting control
- Energy
- Data acquisition
- System management

4.3 Embedded System

Embedded System allows main *System* to communicate with hardware part (*Peripherals*) of micro-based device through *Router* service that resides on *Network Layer* of *Network Model*.

4.3.1 Peripherals

Embedded System supports integration with these Peripherals:

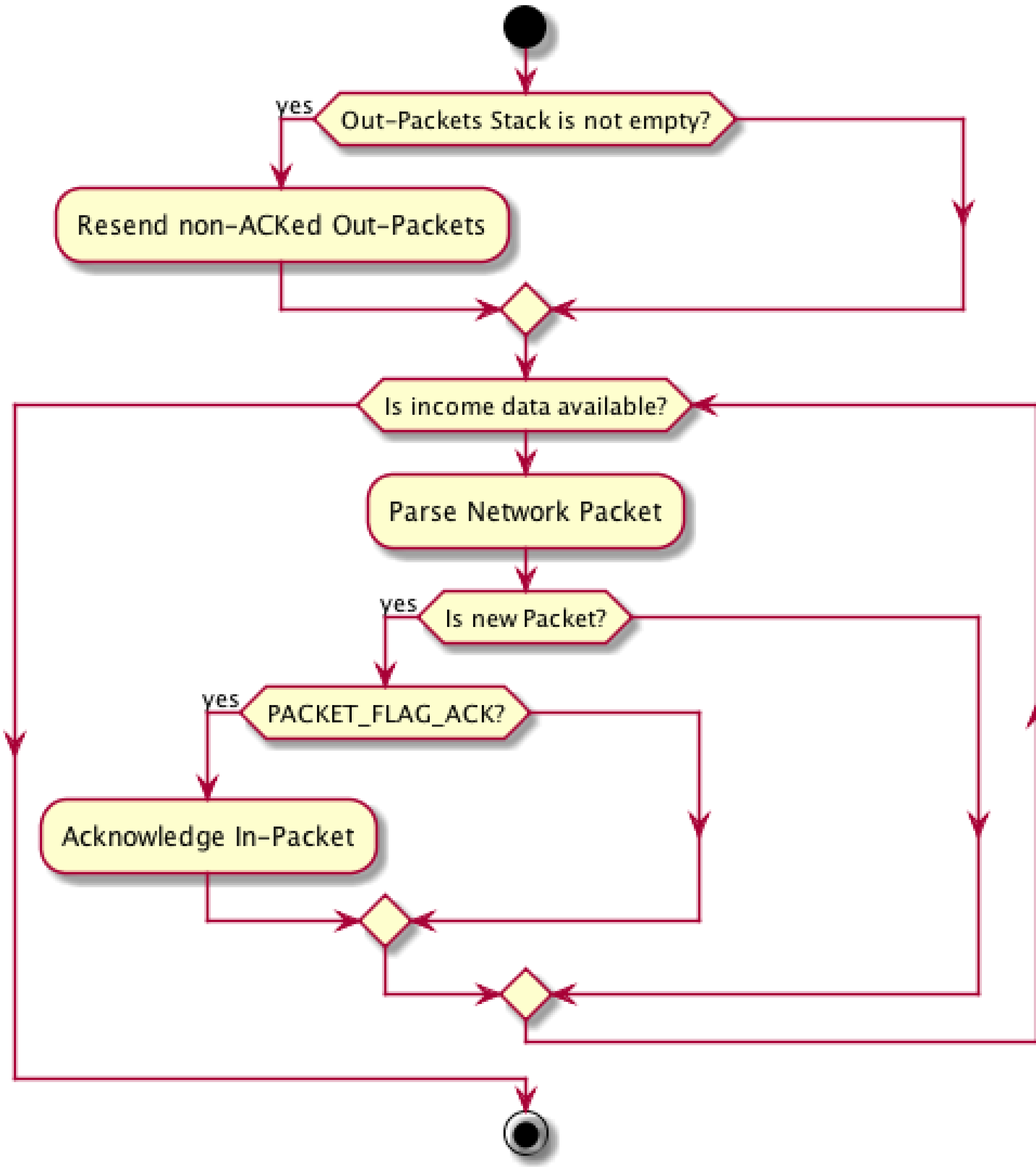
- Serial Communication Interfaces (SCI): RS-232.
- Synchronous Serial Communication Interface: I2C, SPI, 1-Wire
- Networks: Ethernet
- Fieldbuses: CAN.
- Timers
- General Purpose Input/Output (GPIO)
- Analog to Digital/Digital to Analog Convertors (ADC / DAC)

4.3.2 Router

The *Router* service resides on *Network Layer* of *Network Model*. It operates with *Packet* structures and performs the next tasks:

- Parsing of incoming *Packet* from "bytes flow"
- Acknowledging of incoming *Packet* if it has `PACKET_FLAG_ACK`
- Sending an outgoing *Packet*
- Operating with Stack of outgoing *Packets*

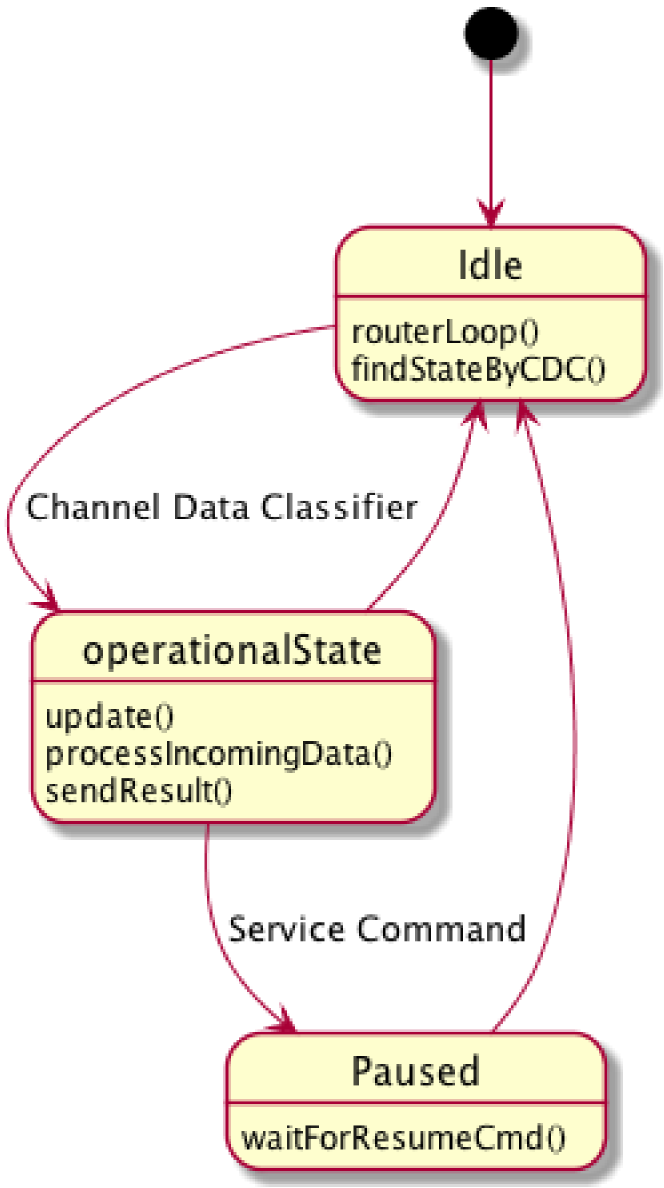
Activity Diagram



4.3.3 Operational State Machine

The *Operational State Machine* is a Finite State Machine with predefined operational states. It can be in only one operational state at a time. The transition from one operational state to another can be initiated by a *Triggering Event* (device interrupt) or *Condition* (based on *Channel Data Classifier*).

State Diagram



Operational States

- *SegmentAcknowledgment*
- *ConfigurePinMode*
- *ReadDigitalPin*
- *WriteDigitalPin*