PSCodec Documentation

Release 0.0.1

Malcolm Mackay

Apr 30, 2021

SPECIFICATION

CHAPTER

ONE

REQUIREMENTS

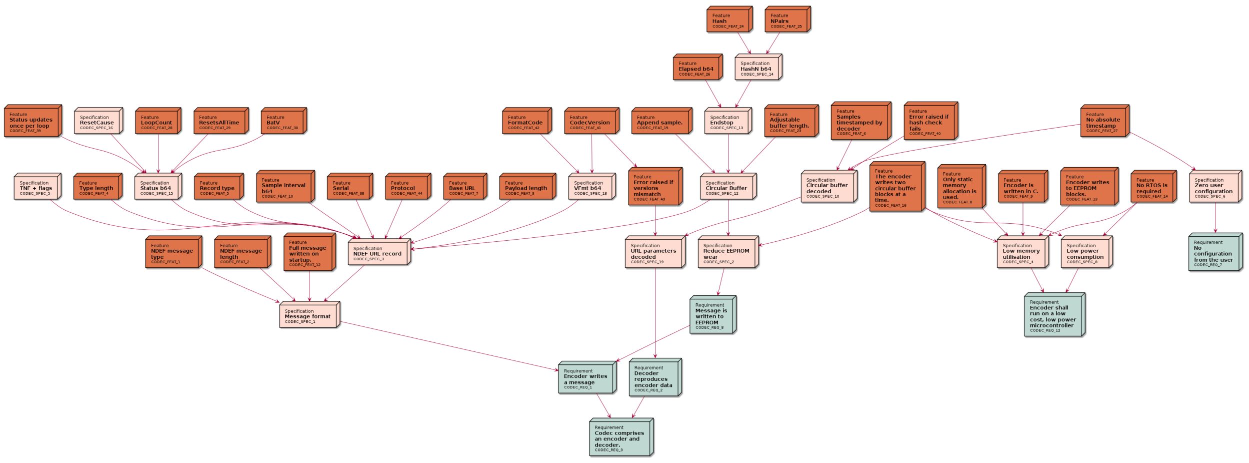

Feature Hash CODEC_FEAT_24
Feature NPairs CODEC_FEAT_25
Feature Elapsed b64 CODEC_FEAT_26
Specification HashN b64 CODEC_SPEC_14
Feature Status updates once per loop CODEC_FEAT_39
Specification ResetCause CODEC_SPEC_16
Feature LoopCount CODEC_FEAT_28
Feature ResetsAllTime CODEC_FEAT_29
Feature BatV CODEC_FEAT_30
Feature FormatCode CODEC_FEAT_42
Feature CodecVersion CODEC_FEAT_41
Feature Append sample. CODEC_FEAT_15
Specification Endstop CODEC_SPEC_13
Feature Adjustable buffer length. CODEC_FEAT_23
Feature Samples timestamped by decoder CODEC_FEAT_6
Feature Error raised if hash check fails CODEC_FEAT_40
Feature No absolute timestamp CODEC_FEAT_27
Specification TNF + flags CODEC_SPEC_5
Feature Type length CODEC_FEAT_4
Specification Status b64 CODEC_SPEC_15
Feature Record type CODEC_FEAT_5
Feature Sample interval b64 CODEC_FEAT_10
Feature Serial CODEC_FEAT_38
Feature Protocol CODEC_FEAT_44
Feature Base URL CODEC_FEAT_7
Feature Payload length CODEC_FEAT_3
Specification VFmt b64 CODEC_SPEC_18
Feature Error raised if versions mismatch CODEC_FEAT_43
Specification Circular Buffer CODEC_SPEC_12
Specification Circular buffer decoded CODEC_SPEC_10
Feature The encoder writes two circular buffer blocks at a time. CODEC_FEAT_16
Feature Only static memory allocation is used. CODEC_FEAT_8
Feature Encoder is written in C. CODEC_FEAT_9
Feature Encoder writes to EEPROM blocks. CODEC_FEAT_13
Feature No RTOS is required CODEC_FEAT_14
Specification Zero user configuration CODEC_SPEC_6
Feature NDEF message type CODEC_FEAT_1
Feature NDEF message length CODEC_FEAT_2
Feature Full message written on startup. CODEC_FEAT_12
Specification NDEF URL record CODEC_SPEC_3
Specification URL parameters decoded CODEC_SPEC_19
Specification Reduce EEPROM wear CODEC_SPEC_2
Specification Low memory utilisation CODEC_SPEC_4
Specification Low power consumption CODEC_SPEC_8
Requirement No configuration from the user CODEC_REQ_7
Specification Message format CODEC_SPEC_1
Requirement Message is written to EEPROM CODEC_REQ_8
Requirement Encoder shall run on a low cost, low power microcontroller CODEC_REQ_12
Requirement Encoder writes a message CODEC_REQ_1
Requirement Decoder reproduces encoder data CODEC_REQ_2
Requirement Codec comprises an encoder and decoder. CODEC_REQ_3

Fig. 1: My first needflow

Requirement: **Codec comprises an encoder and decoder.** *CODEC_REQ_3*
status: open links incoming: *CODEC_REQ_1*, *CODEC_REQ_2*
The codec comprises has two parts: 1. An encoder that produces an URL from raw data. 2. A decoder that recovers raw data from the URL.

1.1 Encoder

Requirement: **Encoder writes a message** *CODEC_REQ_1*

status: open

links outgoing: *CODEC_REQ_3*
links incoming: *CODEC_REQ_8*, *CODEC_SPEC_1*

The encoder takes environmental sensor data and writes it into a message that is opened and read automatically by most mobile phones.

Requirement: **Encoder shall run on a low cost, low power microcontroller** *CODEC_REQ_12*

status: open

links incoming: *CODEC_SPEC_4*, *CODEC_SPEC_8*

The encoder will run on an inexpensive microcontroller. This will be powered by a coin cell battery and should run for years.

Requirement: **No configuration from the user** *CODEC_REQ_7*

status: complete

links incoming: *CODEC_SPEC_6*

The encoder must not require any set up or configuration from the user.

Requirement: **Message is written to EEPROM** *CODEC_REQ_8*
status: complete links outgoing: *CODEC_REQ_1* links incoming: *CODEC_SPEC_2*
The encoder must not write to the same EEPROM block too frequently. Each has a write endurance of roughly 100,000 cycles. Status information changes infrequently compared to environmental sensor data.

1.2 Decoder

Requirement: **Decoder reproduces encoder data** *CODEC_REQ_2*
status: open links outgoing: *CODEC_REQ_3* links incoming: *CODEC_SPEC_19*
The decoder must reproduce data fed into the encoder.

CHAPTER

TWO

SPECIFICATIONS

Specification: **Message format** *CODEC_SPEC_1*

status: complete

links outgoing: *CODEC_REQ_1*
links incoming: *CODEC_FEAT_1*, *CODEC_FEAT_2*, *CODEC_FEAT_12*, *CODEC_SPEC_3*

The message format is NDEF. This is used to transmit data to a phone using NFC. An NDEF message has 3 fields: Type, Length and Value.

NDEF Msg.	NDEF message type (CODEC_FEAT_1)	NDEF message length (CODEC_FEAT_2)			Value
Byte	0	1	2	3	4...
Data	0x03	0xFF	MSB	LSB	NDEF URL record (CODEC_SPEC_3)

Specification: **NDEF URL record** *CODEC_SPEC_3*

status: complete

links outgoing: *CODEC_SPEC_1*

links incoming: *CODEC_FEAT_3*, *CODEC_FEAT_4*, *CODEC_FEAT_5*, *CODEC_FEAT_10*, *CODEC_FEAT_38*, *CODEC_FEAT_44*, *CODEC_FEAT_7*, *CODEC_SPEC_12*, *CODEC_SPEC_18*, *CODEC_SPEC_15*, *CODEC_SPEC_5*

Sensor data are stored in a URL record. As it is the only one in the message and of a known type, a phone opens the URL automatically in its default web browser.

NDEF record header

Desc	TNF + flags (CODEC_SPEC_5)	Type length (CODEC_FEAT_4)	Payload length (CODEC_FEAT_3)				Record type (CODEC_FEAT_5)
Byte	0	1	2	3	4	5	6
Data	0xC3	0x01	PL[3]	PL[2]	PL[1]	PL[0]	0x55

NDEF record payload start

Desc.	Protocol (CODEC_FEAT_44)	Base URL (CODEC_FEAT_7)	Sample interval b64 (CODEC_FEAT_10)	Serial (CODEC_FEAT_38)
Data	0x03	t.plotsensor.com	/?t=AWg*	&s=YWJjZGVm

NDEF record payload continued

Desc.	VFmt b64 (CODEC_SPEC_18)	Status b64 (CODEC_SPEC_15)	CircBuffer-Start	Circular Buffer (CODEC_SPEC_12)
Data	&v=AAAA	&x=AAABALEK	&q=	MDaWMDaW…

Specification: **Circular Buffer** *CODEC_SPEC_12*

status: complete

links outgoing: *CODEC_SPEC_3*, *CODEC_SPEC_2*
links incoming: *CODEC_FEAT_23*, *CODEC_FEAT_15*, *CODEC_SPEC_13*

The circular buffer starts on a block boundary and occupies an integer number of 16-byte blocks. 1K of EEPROM is enough for 32 blocks.
Only two blocks are edited in RAM at a time:

Cursor Block					Next Block				
Cursor Demi				Endstop Demis (0,1)		Oldest Demi			
$P_{64}1$		$P_{64}0$				$P_{64}N$		$P_{64}N-1$	
$R_{64}3$	$R_{64}2$	$R_{64}1$	$R_{64}0$			$R_{64}L$	$R_{64}L-1$	$R_{64}L-2$	$R_{64}L-3$

Blocks are subdivided into two 8-byte demis. Each demi holds 2 base64 encoded pairs.
Each pair consists of 2 base64 encoded sensor readings. By default these will be captured simultaneously by a temperature sensor and a humidity sensor.
New sensor readings are written to Cursor Demi. Each time this occurs, the subsequent Endstop (CODEC_SPEC_13) is updated.
When Cursor Demi is full, both it and the endstop are moved forward when the next sensor reading is added:

Cursor Block							Next Block
Demi				Cursor Demi			Endstop Demis (0,1)
S2		S1		S0		Spad	
R5	R4	R3	R2	R1	R0		

The previous oldest demi is overwritten. Note there can be a gap between the most recent sample and the start of the endstop demis. This is zero padded. The padding will not be decoded because the number of valid samples in the buffer is included in the endstop.

Specification: **Endstop** *CODEC_SPEC_13*

status: complete

links outgoing: *CODEC_SPEC_12*
links incoming: *CODEC_FEAT_26*, *CODEC_SPEC_14*

The endstop occupies 2 demis (16 bytes) after the cursor demi. It is terminated with a unique character. This marks the end of the circular buffer; the divide between new and old data. The decoder finds this in order to unwrap the circular buffer into a list of samples, ordered newest to oldest.
The endstop contains data about the current state of the circular buffer, for example the number of valid samples it contains. These data are appended to the circular buffer to meet Reduce EEPROM wear (CODEC_SPEC_2).

Specification: **VFmt b64** *CODEC_SPEC_18*

status: open

links outgoing: *CODEC_SPEC_3*
links incoming: *CODEC_FEAT_41*, *CODEC_FEAT_42*

This is a 3 byte structure that expands to 4 bytes after base64 encoding.
The unencoded structure is:

Byte	0	1	2
Description	CodecVersion (CODEC_FEAT_41)		FormatCode (CODEC_FEAT_42)

Specification: **HashN b64** *CODEC_SPEC_14*

status: complete

links outgoing: *CODEC_SPEC_13*
links incoming: *CODEC_FEAT_24*, *CODEC_FEAT_25*

This is a 9 byte structure that expands to 12 bytes after base64 encoding.
The unencoded structure is:

Byte	0	1	2	3	4	5	6	7	8
Description	Hash (CODEC_FEAT_24)							NPairs (CODEC_FEAT_25)	

Specification: **Status b64** *CODEC_SPEC_15*

status: complete

links outgoing: *CODEC_SPEC_3*

links incoming: *CODEC_FEAT_28*, *CODEC_FEAT_29*, *CODEC_FEAT_30*, *CODEC_FEAT_39*, *CODEC_SPEC_16*

This is a 6 byte structure that expands to 8 bytes after base64 encoding.
It corresponds to `status`. Status information is used by the decoder to determine if the encoder and its microcontroller host are running ok.
The unencoded structure is:

Byte	0	1	2	3	4	5
Description	LoopCount (CODEC_FEAT_28)		ResetsAllTime (CODEC_FEAT_29)		BatV (CODEC_FEAT_30)	ResetCause (CODEC_SPEC_16)

Specification: **ResetCause** *CODEC_SPEC_16*

status: complete

links outgoing: *CODEC_SPEC_15*

links incoming: *CODEC_FEAT_31*, *CODEC_FEAT_32*, *CODEC_FEAT_33*, *CODEC_FEAT_34*, *CODEC_FEAT_35*, *CODEC_FEAT_36*, *CODEC_FEAT_37*

Flags to indicate causes of the most recent microcontroller reset.

Bit	0	1	2	3	4	5	6	7
Description	BOR (CODEC_FEAT_31)	SVSH (CODEC_FEAT_32)	WDT (CODEC_FEAT_33)	MISC (CODEC_FEAT_34)	LPM5WU (CODEC_FEAT_35)	CLOCKFAIL (CODEC_FEAT_36)		SCANTIMEOUT (CODEC_FEAT_37)

Specification: **TNF + flags** *CODEC_SPEC_5*

status: complete

links outgoing: *CODEC_SPEC_3*

links incoming: *CODEC_FEAT_17*, *CODEC_FEAT_18*, *CODEC_FEAT_19*, *CODEC_FEAT_20*, *CODEC_FEAT_21*, *CODEC_FEAT_22*

TNF and flags for the NDEF record.

Bit	7	6	5	4	3	2	1	0
Field	MB (CODEC_FEAT_17)	ME (CODEC_FEAT_18)	CF (CODEC_FEAT_19)	SR (CODEC_FEAT_20)	IL (CODEC_FEAT_21)	TNF (CODEC_FEAT_22)		
Data	1	1	0	0	0	0x03		

Specification: **Low memory utilisation** *CODEC_SPEC_4*

status: open

links outgoing: *CODEC_REQ_12*

links incoming: *CODEC_FEAT_13*, *CODEC_FEAT_8*, *CODEC_FEAT_9*, *CODEC_FEAT_14*, *CODEC_FEAT_16*

The encoder must use <2K of RAM and <16K of non-volatile FRAM, as can be found on an MSP430FR2033 microcontroller.

Specification: **Reduce EEPROM wear** *CODEC_SPEC_2*

status: open

links outgoing: *CODEC_REQ_8*

links incoming: *CODEC_FEAT_16*, *CODEC_SPEC_12*

Specification: **Low power consumption** *CODEC_SPEC_8*
status: open links outgoing: *CODEC_REQ_12* links incoming: *CODEC_FEAT_14*, *CODEC_FEAT_16*
The encoder will run for >2 years on a hardware powered by a CR1620 battery.

Specification: **Zero user configuration** *CODEC_SPEC_6*
links outgoing: *CODEC_REQ_7* links incoming: *CODEC_FEAT_27*
The encoder must run without input from the user. This includes after the Power-on-Reset when a battery is replaced.

Specification: **URL parameters decoded** *CODEC_SPEC_19*
links outgoing: *CODEC_REQ_2* links incoming: *CODEC_FEAT_43*, *CODEC_SPEC_10*
Before the circular buffer is decoded, URL parameters such as VFmt b64 (CODEC_SPEC_18) are needed.

Specification: **Circular buffer decoded** *CODEC_SPEC_10*
links outgoing: *CODEC_SPEC_19* links incoming: *CODEC_FEAT_40*, *CODEC_FEAT_27*, *CODEC_FEAT_6*
The decoder outputs a list of samples from the URL. Output depends on FormatCode (CODEC_FEAT_42). By default samples will contain temperature and humidity readings, converted to degrees C and percent respectively. Each will have a timestamp precise to one minute. This corresponds to the time that the sample was added to the circular buffer.

CHAPTER

THREE

FEATURES

3.1 NDEF message

Feature: **NDEF message type** *CODEC_FEAT_1*
links outgoing: *CODEC_SPEC_1*
The message type is 0x03, corresponding to a known type.

Feature: **NDEF message length** *CODEC_FEAT_2*
links outgoing: *CODEC_SPEC_1*
Message length in bytes as a 16-bit value. Byte 2 is unused so 0xFF. Byte 1 holds the Most Significant 8-bits. Byte 0 holds the Least Significant 8 bits. There is no function to change this after the message has been created.

3.1.1 NDEF record

Feature: **Payload length** *CODEC_FEAT_3*
status: complete links outgoing: *CODEC_SPEC_3*
Length of the NDEF record payload length in bytes. Similar to NDEF message length (CODEC_FEAT_2), it cannot change after the record has been created.

Feature: **Type length** *CODEC_FEAT_4*
status: complete links outgoing: *CODEC_SPEC_3*
Length of the Record type (CODEC_FEAT_5) field in bytes. This is 1 byte.

Feature: **Record type** *CODEC_FEAT_5*
status: complete links outgoing: *CODEC_SPEC_3*
NDEF record type is 0x55, which corresponds to a URI record.

URL Parameters

Feature: **Sample interval b64** *CODEC_FEAT_10*
status: complete links outgoing: *CODEC_SPEC_3* links incoming: *CODEC_IMPL_1*
The time interval between samples in minutes. This must be constant.

Feature: **Serial** *CODEC_FEAT_38*
status: complete links outgoing: *CODEC_SPEC_3* links incoming: *CODEC_IMPL_8*
An 8 character serial string uniquely identifies the encoder instance. More generally this will identify the hardware that the encoder is running on. Characters from the base64 dictionary are recommended for these are URL safe.

Feature: **CodecVersion** *CODEC_FEAT_41*
status: open links outgoing: *CODEC_SPEC_18*, *CODEC_FEAT_43*
16-bit unsigned integer codec version. From this the decoder can raise an error if it is not compatible.

Feature: **FormatCode** *CODEC_FEAT_42*

status: open

links outgoing: *CODEC_SPEC_18*

8-bit identifier of the circular buffer format. The circular buffer is arranged into pairs. A sample either corresponds to a pair of readings (e.g. temperature and humidity), or a single reading (temperature only). The latter option doubles the number of samples in the buffer.

The device is specified as HDC2021. This allows the decoder to convert from the sensor ADC value (a 12-bit integer) into floating point degrees C or percent. Equations to do this are device specific.

FormatCode	Definition
0	HDC2021_TRH_OnePairPerSample
1	HDC2021_T_OneReadingPerSample

Feature: **Error raised if versions mismatch** *CODEC_FEAT_43*

status: open

links outgoing: *CODEC_SPEC_19*
links incoming: *CODEC_FEAT_41*

If the decoder version does not match that of the encoder used to produce the URL, then Decoder reproduces encoder ... (CODEC_REQ_2) cannot be guaranteed.

Feature: **Protocol** *CODEC_FEAT_44*

status: open

links outgoing: *CODEC_SPEC_3*

The HTTPS protocol is recommended for production use.

Protocol	Definition
0x03	http://
0x04	https://

Status

Feature: **LoopCount** *CODEC_FEAT_28*
status: complete links outgoing: *CODEC_SPEC_15*
The number of times the circular buffer has looped from the last EEPROM block to the first since initialisation. See `loopcount`.

Feature: **ResetsAllTime** *CODEC_FEAT_29*
status: complete links outgoing: *CODEC_SPEC_15*
Number of times the microcontroller running the encoder has reset. Each reset causes a counter to be incremented in non-volatile memory (`resetsalltime`).

Feature: **BatV** *CODEC_FEAT_30*
status: complete links outgoing: *CODEC_SPEC_15*
The battery voltage in mV. See `batvoltage`.

ResetCause

Feature: **BOR** *CODEC_FEAT_31*
status: complete tags: bit links outgoing: *CODEC_SPEC_16*
Brown Out Reset flag.

Feature: **SVSH** *CODEC_FEAT_32*
status: complete tags: bit links outgoing: *CODEC_SPEC_16*
Supply Voltage Supervisor error flag.

Feature: **WDT** *CODEC_FEAT_33*
status: complete tags: bit links outgoing: *CODEC_SPEC_16*
Watchdog Timeout flag

Feature: **MISC** *CODEC_FEAT_34*

status: complete
tags: bit

links outgoing: *CODEC_SPEC_16*

Miscellaneous Error flag

Feature: **LPM5WU** *CODEC_FEAT_35*

status: complete
tags: bit

links outgoing: *CODEC_SPEC_16*

Low Power Mode x.5 wakeup flag.

Feature: **CLOCKFAIL** *CODEC_FEAT_36*

status: complete
tags: bit

links outgoing: *CODEC_SPEC_16*

Clock failure flag.

Feature: **SCANTIMEOUT** *CODEC_FEAT_37*

status: complete
tags: bit

links outgoing: *CODEC_SPEC_16*

Scan timeout flag.

Circular Buffer

Feature: **Error raised if hash check fails** *CODEC_FEAT_40*

status: complete

links outgoing: *CODEC_SPEC_10*
links incoming: *CODEC_IMPL_5*

The decoder independently calculates the hash of the circular buffer and compares it with the one contained in Endstop (CODEC_SPEC_13). If the check fails then no samples are returned and an exception is raised.
If the MD5 hash is used then this indicates the decoded sample list does not correspond to that fed into the encoder. Therefore Decoder reproduces encoder ... (CODEC_REQ_2) has not been met.
If the HMAC hash is used then there is an additional possibility: authentication has failed. The secret key used by the encoder and the stored copy used by the decoder do not match. This occurs when the software is run by an unauthorised 3rd party.

Feature: **Adjustable buffer length.** *CODEC_FEAT_23*

status: complete

links outgoing: *CODEC_SPEC_12*
links incoming: *CODEC_IMPL_3*

The length of the circular buffer can be adjusted. This is done with a compiler parameter, to meet Only static memory allocati... (CODEC_FEAT_8).

Feature: **Hash** *CODEC_FEAT_24*

status: complete

links outgoing: *CODEC_SPEC_14*
links incoming: *CODEC_IMPL_5*

The list of samples in the buffer must always be transmitted together with a hash. This is used by the decoder to verify that it has unwrapped the circular buffer and decoded samples correctly.
The size of the URL is limited, so there is only room to store the least significant 7 bytes of the hash, however, this should be ample. The hash does not contain Elapsed b64 (CODEC_FEAT_26) and therefore it does not need to be recalculated each time this changes. This is done in order to save power Low power consumption (CODEC_SPEC_8).
If Hash Based Message Authentication (HMAC) is enabled, then the last characters of the HMAC-MD5 will be used. If not, these will be the output of MD5 only.
The hash is used as a checksum; a guard against unintentional data corruption. This may arise because of a bug in the codec. The MD5 is sufficient for this purpose. It was chosen in order to adhere with Low memory utilisation (CODEC_SPEC_4). The MD5 hash alone is useless for security purposes. If a bad actor intends to find a collision (i.e. two sets of data that produce the same MD5 hash) then this can be done with ease. MD5 is intended for debug and code development only.
HMAC-MD5 is considerably more secure than MD5 alone. It is recommended for production use. Each device with an encoder should have a unique secret key. In addition to data integrity, HMAC-MD5 can be used to verify that the decoder and encoder have access to the same shared secret key. It is therefore a check on authenticity.

> **From Wikipedia:** The cryptographic strength of the HMAC depends upon the size of the secret key that is used. The most common attack against HMACs is brute force to uncover the secret key. HMACs are substantially less affected by collisions than their underlying hashing algorithms alone.[6][7] In particular, in 2006 Mihir Bellare proved that HMAC is a PRF under the sole assumption that the compression function is a PRF.[8] Therefore, HMAC-MD5 does not suffer from the same weaknesses that have been found in MD5.
>
> . . .
>
> For HMAC-MD5 the RFC summarizes that – although the security of the MD5 hash function itself is severely compromised – the currently known "attacks on HMAC-MD5 do not seem to indicate a practical vulnerability when used as a message authentication code", but it also adds that "for a new protocol design, a ciphersuite with HMAC-MD5 should not be included".

It is acknowledged that HMAC-MD5 has been used despite the counter-recommendation above. I decided that the increased complexity of HMAC-SHA3 cannot be justified: The algorithm has to run with low energy consumption on an inexpensive microcontroller (see Encoder shall run on a low ... (CODEC_REQ_12)). The MSP430 itself is not designed for a high degree of data security. De-lidding and X-raying are possible. This is why it is important not to share the secret key between devices. Opting for a more robust hashing algorithm may result in compromises elsewhere (e.g. on battery life). It is also the case that environmental sensor data are being transmitted and not data that are highly sensitive. The reward to compromise this system is sufficiently low to make HMAC-MD5 a good-enough deterrent.
Data are hashed in the following order:

Byte	Field	Value
0	Pair[0]	Reading0_MSB
1		Reading1_MSB
2		LSB
3	Pair[1]	Reading0_MSB
4		Reading1_MSB
5		LSB
. . .	. . .	
L-11	Pair[NPairs-1]	Reading0_MSB
L-10		Reading1_MSB
L-9		LSB

Feature: **NPairs** *CODEC_FEAT_25*
status: complete links outgoing: *CODEC_SPEC_14* links incoming: *CODEC_IMPL_4*
Number of valid samples in the circular buffer. This excludes samples used for padding. Populated from `npairs`.

Feature: **Elapsed b64** *CODEC_FEAT_26*
status: complete links outgoing: *CODEC_SPEC_13* links incoming: *CODEC_IMPL_2*
External to the codec is a counter. This increases by 1 every minute after the previous sample was written to the circular buffer. It resets to 0 when a new sample is written. The decoder uses it to determine to the nearest minute when samples were collected. Without it, the maximum resolution on the timestamp for each sample would be equal to the time interval, which can be up to 60 minutes. The unencoded minutes elapsed field is 16-bits wide. This is the same width as the unencoded time interval in minutes field. The minutes elapsed field occupies 4 bytes after base64 encoding, including one padding byte. By convention this is 0x61 or '='. The encoder replaces the padding byte with `ENDSTOP_BYTE`. This marks the last byte of the end stop. The first step performed by the decoder is to locate `ENDSTOP_BYTE`. After it is found, it can be replaced with an '=' before the minutes elapsed field is decoded from base64 into its original 16-bit value.

Flags + TNF

Feature: **MB** *CODEC_FEAT_17*
status: complete tags: bit links outgoing: *CODEC_SPEC_5*
Message Begin bit denotes the first record in an NDEF message. This is set. The record is the first in the message.

Feature: **ME** *CODEC_FEAT_18*
status: complete tags: bit links outgoing: *CODEC_SPEC_5*
Message End bit denotes the last record in an NDEF message. This is set. The record is the last in the message.

Feature: **CF** *CODEC_FEAT_19*
status: complete tags: bit links outgoing: *CODEC_SPEC_5*
Chunk Flag bit denotes a message comprised of several records chunked together (concatenated). This is cleared. There is only one record in the message.

Feature: **SR** *CODEC_FEAT_20*
status: complete tags: bit links outgoing: *CODEC_SPEC_5*
Short Record bit. When set Payload length (CODEC_FEAT_3) one byte long. When cleared it is 4 bytes long. This is cleared, because the message is longer than 255 bytes.

Feature: **IL** *CODEC_FEAT_21*
status: complete tags: bit links outgoing: *CODEC_SPEC_5*
ID Length bit. When set the ID length field is present. When cleared it is omitted. This is cleared.

Feature: **TNF** *CODEC_FEAT_22*
status: complete tags: bit links outgoing: *CODEC_SPEC_5*
Type Name Format field. A 3-bit value that describes the record type. This is set to 0x03, which corresponds to an Absolute URI Record.

3.2 Other

Feature: **No absolute timestamp** *CODEC_FEAT_27*
links outgoing: *CODEC_SPEC_6*, *CODEC_SPEC_10*
The URL from the encoder cannot include an absolute timestamp. This would need to be set each time the microcontroller is powered on (e.g. when the battery is replaced).

Feature: **Samples timestamped by decoder** *CODEC_FEAT_6*
links outgoing: *CODEC_SPEC_10* links incoming: *CODEC_IMPL_1*
All samples are timestamped relative to the time that the decoder is run. It is assumed that the time difference between when the encoded message is read (by a phone) and the time the decoder is run (on a web server) is much less than one minute. Timestamp precision is one minute. The timestamping algorithm is as follows: #. Samples are put in order of recency. #. Minutes Elapsed b64 (CODEC_FEAT_26) since the most recent sample is extracted from the URL. #. Current time (now in UTC) is determined. #. The first sample is assigned a timestamp = now - minutes elapsed. #. Sample interval b64 (CODEC_FEAT_10) between samples is extracted from the URL. This is used to timestamp each sample relative to the first.

Feature: **Base URL** *CODEC_FEAT_7*
links outgoing: *CODEC_SPEC_3*
The base URL can be changed. It is recommended to keep this as short as possible to allow more room for environmental sensor data.

3.3 Low resource utilisation

Feature: **Encoder writes to EEPROM blocks.** *CODEC_FEAT_13*
status: complete links outgoing: *CODEC_SPEC_4*
The encoder cannot output the 1000 character NDEF message in one go. This would require too much RAM for a small microcontroller. Instead it is designed to output an I2C EEPROM, which is arranged into 16-byte blocks. A maximum of 4 EEPROM blocks are written to or read from at a time.

Feature: **Only static memory allocation is used.** *CODEC_FEAT_8*
status: complete links outgoing: *CODEC_SPEC_4*
The stdio library needed for malloc takes a lot of available memory on the MSP430, so it is not used.

Feature: **Encoder is written in C.** *CODEC_FEAT_9*
status: complete links outgoing: *CODEC_SPEC_4*
There is little benefit to C++ given the low complexity of the encoder.

Feature: **No RTOS is required** *CODEC_FEAT_14*
status: complete links outgoing: *CODEC_SPEC_8*, *CODEC_SPEC_4*
An RTOS is not appropriate for this application. It will significantly increase the memory footprint. It will add complexity and make power consumption more difficult to control.

Feature: **Status updates once per loop** *CODEC_FEAT_39*
status: complete links outgoing: *CODEC_SPEC_15*
Status contains some parameters that change infrequently. For these, Reduce EEPROM wear (CODEC_SPEC_2) is not a concern. LoopCount (CODEC_FEAT_28) and BatV (CODEC_FEAT_30) are updated once, when cursorblk and nextblk are at opposite ends of the circular buffer (e.g. cursorblk == 31 and nextblk == 0). This will happen once per day. ResetCause (CODEC_SPEC_16) is cleared when this happens, because a reset has not occurred recently.

Feature: **Full message written on startup.** *CODEC_FEAT_12*
status: complete links outgoing: *CODEC_SPEC_1* links incoming: *CODEC_IMPL_7*
The entire NDEF message only needs to be written once upon startup. Afterwards, small parts of the message are modified at a time.

Feature: **Append sample.** *CODEC_FEAT_15*
status: complete links outgoing: *CODEC_SPEC_12* links incoming: *CODEC_IMPL_6*
The list of environmental sensor readings (and its HMAC) will change at an interval of time interval minutes. If the time interval is set to 5 minutes, 100K writes will be reached in (5 minutes * 100e3) = 1 year. By using a circular buffer, these writes are distributed across many blocks. This is a form of *Wear levelling <https://en.wikipedia.org/wiki/Wear_leveling>*.

Feature: **The encoder writes two circular buffer blocks at a time.** *CODEC_FEAT_16*
status: complete links outgoing: *CODEC_SPEC_4*, *CODEC_SPEC_2*, *CODEC_SPEC_8*
This reduces the requirement for RAM on the MSP430 and reduces power consumption (it takes time to write EEPROM blocks).

CHAPTER

FOUR

IMPLEMENTATION

Implementation: **Sample interval** *CODEC_IMPL_1*
status: complete links outgoing: *CODEC_FEAT_10*, *CODEC_FEAT_6*
The time interval between samples (in minutes) is defined in the global variable `smplintervalmins`. `ndef_writepreamble()` converts `smplintervalmins` into a base64 string and writes it to URL parameter ‘t’. Decoder method `decode_timeinterval` converts this back to an integer.

Implementation: **Elapsed time** *CODEC_IMPL_2*
status: complete links outgoing: *CODEC_FEAT_26*
The function `enc_setelapsed()` alters the elapsed time field, independent of the rest of the URL. It is intended that this is called once for each minute after a sample is taken. Elapsed time (as an integer) is converted to base64 and written to the end stop. Base64 elapsed time is extracted in `BufferDecoder` and converted back to an integer.

Implementation: **Length in blocks** *CODEC_IMPL_3*
status: complete links outgoing: *CODEC_FEAT_23*
Buffer length is set at compile time with `BUFSIZE_BLKS`.

Implementation: **Length in samples** *CODEC_IMPL_4*
status: complete links outgoing: *CODEC_FEAT_25*
The function `enc_pushsample()` uses integer `npairs` to record how many valid samples are in the circular buffer. When an demi is overwritten, it is reduced by `PAIRS_PER_DEMI`. Otherwise it is incremented by one. When the buffer is full `npairs` will equal `buflenpairs`.

Implementation: **MD5** *CODEC_IMPL_5*
status: complete links outgoing: *CODEC_FEAT_24*, *CODEC_FEAT_40*
The encoder maintains `pairhistory`, a RAM-based shadow of the EEPROM circular buffer. It consumes a lot of RAM, but this is unavoidable. On each call to `enc_pushsample()`, the sample is appended to `pairhistory` by `pairhist_push()`. The hash (MD5 or HMAC) is calculated with `pairhist_hash()`. This outputs a 9 byte structure (`hashn_t`). It is converted to base64 (`hashnb64`) before it is written to the endstop demis (Endstop (CODEC_SPEC_13)).

Implementation: **Append sample** *CODEC_IMPL_6*
status: complete links outgoing: *CODEC_FEAT_15*
Samples are added to the circular buffer with *`enc_pushsample()`*. This takes one or two measurands, depending on the circular buffer format.

Implementation: **Initialisation** *CODEC_IMPL_7*
status: complete links outgoing: *CODEC_FEAT_12*
The NDEF message and its circular buffer are initialised with *`enc_init()`*. Given there are no samples in the circular buffer, the endstop and cursor are omitted. All demis are set to MDaW (all zeroes). State machines in the `sample` and `demi` files are reset.

Implementation: **Serial** *CODEC_IMPL_8*
status: complete links outgoing: *CODEC_FEAT_38*
The serial string is defined in the global variable `serial`. This must be *`SERIAL_LENBYTES`* long. It must contain only URL-safe characters. *`ndef_writepreamble()`* copies this into URL parameter ‘s’.

CHAPTER

FIVE

DECODER

5.1 Decode a cuplcodec URL

The decoder extracts a timestamped list of samples from a cuplcodec URL.

`wscodec.decoder.decoderfactory.`**`_get_decoder`**(*formatcode: int*)

Parameters **`formatcode`** – Value of the codec format field. Specifies which decoder shall be returned.

Returns

Return type Decoder class for the given format code.

`wscodec.decoder.decoderfactory.`**`decode`**(*secretkey: str, statb64: str, timeintb64: str, circb64: str, vfmtb64: str, usehmac: bool = True, scantimestamp: Optional[datetime.datetime] = None*) → *wscodec.decoder.samples.SamplesURL*

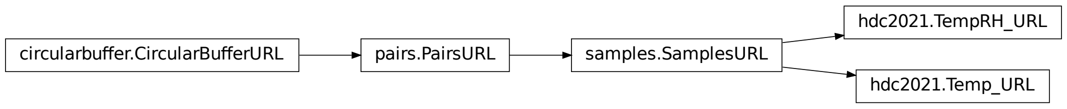
Decode the version string and extract codec version and format code. An error is raised if the codec version does not match. A decoder object is returned based on the format code. An error is raised if no decoder is available for the code.

Parameters

- **`secretkey`** (*`str`*) – HMAC secret key as a string. Normally 16 characters long.
- **`statb64`** (*`str`*) – Value of the URL parameter that holds status information (base64 encoded).
- **`timeintb64`** (*`str`*) – Value of the URL parameter that holds the time interval between samples in minutes (base64 encoded).
- **`circb64`** (*`str`*) – Value of the URL parameter that contains the circular buffer of base64 encoded samples.
- **`vfmtb64`** (*`str`*) – Value of the URL parameter that contains the version and format string (base64 encoded).
- **`usehmac`** (*`bool`*) – True if the hash inside the circular buffer endstop is HMAC-MD5. False if it is MD5.
- **`scantimestamp`** (*`datetime`*) – The time that the tag was scanned. All decoded samples will be timestamped relative to this.

Returns An object containing a list of timestamped environmental sensor samples.

Return type *SamplesURL*



class wscodec.decoder.hdc2021.**TempRH_URL**(**args*, ***kwargs*)

class wscodec.decoder.hdc2021.**Temp_URL**(**args*, ***kwargs*)

class wscodec.decoder.samples.**SamplesURL**(**args*, *timeintb64: str*, *scantimestamp: Optional[datetime.datetime] = None*, ***kwargs*)

This holds a list of decoded sensor samples. Each needs a timestamp, but this must be calculated. There are no absolute timestamps in the URL. First, all times are relative to scantimestamp (when the tag was scanned).

The URL does contain self.elapsedmins (the minutes elapsed since the newest sample was acquired). This makes it possible to calculate self.newest_timestamp, the timestamp of the newest sample.

Every subsequent sample is taken at a fixed time interval relative to self.newest_timestamp. This time interval is decoded from timeintb64 into self.timeinterval.

Parameters

- ***args** – Variable length argument list
- **timeintb64** (*str*) – Time interval between samples in minutes, base64 encoded into a 4 character string.
- **scantimestamp** (*datetime*) – Time the tag was scanned. It corresponds to the time the URL on the tag is requested from the web server.
- ****kwargs** – Keyword arguments to be passed to parent class constructors.

generate_timestamp()

Yields *A timestamp of a sample, calculated relative to that of the newest sample.*

get_samples_list()

Returns

Return type Samples as a list of dictionaries. This is done for compatibility purposes.

class wscodec.decoder.pairs.**PairsURL**(**args*, *usehmac: bool = False*, *secretkey: Optional[str] = None*, ***kwargs*)

This takes the payload of the linearised buffer, which is a long string of base64 characters. It decodes this into a list of pairs. The hash (MD5 or HMAC-MD5) is taken and compared with that supplied in the URL by the encoder. If the hashes match then the decode has been successful. If not, an exception is raised.

Parameters

- ***args** – Variable length argument list.
- **usehmac** (*bool*) – True if the hash inside the circular buffer endstop is HMAC-MD5. False if it is MD5.
- **secretkey** (*str*) – HMAC secret key as a string. Normally 16 characters long.
- ****kwargs** – Keyword arguments to be passed to parent class constructors.

`_decode_pairs``()`

The payload string is converted into a list of 8-byte demis (see demi).

The first demi is the newest; its data have been written to the circular buffer most recently, so it closest to the left of the endstop. It can contain either one or two pairs. This is decoded first.

Subsequent (older) demis each contain 2 pairs. These are decoded. The final list of pairs is in chronological order with the newest first and the oldest last.

`static _dividestring``(`*source: str*, *n: int*`)`

Parameters

- **`source`** (*`str`*) – The string to be divided.
- **`n`** – The number of characters in each substring.

Returns

Return type A list of substrings, each containing n characters.

`static _gethash``(`*message: bytearray*, *usehmac: bool*, *secretkey: str*`)`

Calculates the hash of a message.

Parameters

- **`message`** (*`bytearray`*) – Input data to the hashing algorithm.
- **`usehmac`** (*`bool`*) – When True the HMAC-MD5 algorithm is used. Otherwise MD5 is used (not recommended for production).
- **`secretkey`** (*`str`*) – HMAC secret key as a string. Normally 16 characters long.

Returns

- **digest** (*str*) – The message hash.
- **hashtype** (*HashType*) – The hash algorithm used.

`_pairsfromdemi``(`*demi: str*`)` → List[*wscodec.decoder.pairs.Pair*]

Decode a demi into 2 pairs.

Parameters **`demi`** (*`str`*) – A string containing 8 base64 characters.

Returns Element 0 is the oldest pair, decoded from the first 4 demi characters. Element 1 is the newest pair, decoded from the last 4 demi characters.

Return type A list of 2 pairs

`_verify``(`*usehmac: bool*, *secretkey: str*`)`

Calculate a hash from the list of pairs according to the same algorithm used by the encoder (see pairhist_hash). Besides pairs, data from the status URL parameter are included. This makes it very unlikely that the same data will be hashed twice, as well as 'protecting' the status parameter from modification by a 3rd party.

A fragment of the calculated hash is compared with that supplied by the encoder. If the hashes agree then verification is successful. If not, an exception is raised.

Parameters

- **`usehmac`** (*`bool`*) – True if the hash inside the circular buffer endstop is HMAC-MD5. False if it is MD5.
- **`secretkey`** (*`str`*) – HMAC secret key as a string. Normally 16 characters long.

Raises **`MessageIntegrityError`** – If the hash calculated by this decoder does not match the hash provided by the encoder.:

class `wscodec.decoder.circularbuffer.`**`CircularBufferURL`**(*statb64: str, circb64: Optional[str] = None*)

Base class for a cuplcodec URL.

This includes at least a circular buffer with a long string of base64 encoded sample data and a short status field.

Instantiation decodes the status string first. It contains error information from the microcontroller running the encoder.

Next it locates the ENDSTOP_BYTE in the circular buffer string. Characters to its left are the newest. Characters to its right are the oldest. The circular buffer is unwrapped into a string where ENDSTOP_BYTE is the last character and the oldest data is in the first.

The linearised buffer is further divided into two parts: The endstop string (including the endstop itself) are at the end. It contains metadata such as the number of samples in the payload. This is preceded by the payload string, which contains a list base64-encocded environmental sensor readings. These are in chronological order oldest-to-newest reading left-to-right.

The decoding of the payload string is handled elsewhere.

Parameters

statb64 [str] Base64 encoded status string extract from a URL parameter.

circb64 [str] A long string containing base64 encoded samples that are organised as a circular buffer.

`ELAPSED_LEN_BYTES = 4`

Length of the endstop elapsed minutes field in bytes (including the endstop itself).

`ENDSTOP_BYTE = '~'`

The last character in the endstop and the end of the circular buffer. Must be URL safe.

`ENDSTOP_LEN_BYTES = 16`

Length of the endstop in bytes.

`_decode_endstop``()`

Decode the circular buffer endstop. This can be over-ridden by a child of this class if the endstop data needs to change in future.

`_decode_status``()`

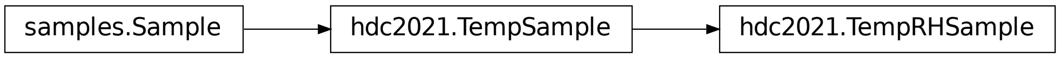
Instantiate a Status object. This can be over-ridden by a child of this class if the Status data needs to change in future.

`_linearise``()`

Linearise the circular buffer.

The circular buffer is made linear by concatenating the two parts of the buffer either side of the end stop.

5.2 Sample



class wscodec.decoder.hdc2021.**TempRHSample**(*rawtemp: int*, *rawrh: int*, *timestamp: datetime.datetime*)

static reading_to_rh(*reading: int*) → float

Parameters **reading** – Integer Relative Humidity ADC reading from the HDC2021.

Returns Relative Humidity in percent.

class wscodec.decoder.hdc2021.**TempSample**(*rawtemp: int*, *timestamp: datetime.datetime*)

static reading_to_temp(*reading: int*) → float

Parameters **reading** – Integer temperature ADC reading from the HDC2021.

Returns Temperature in degrees C

class wscodec.decoder.samples.**Sample**(*timestamp: datetime.datetime*)

Sample base class. All samples must contain a timestamp.

5.3 Pair

class wscodec.decoder.pairs.**Pair**(*rd0MSB: int*, *rd1MSB: int*, *Lsb: int*)

Class representing a pair of 12-bit sensor readings.

In the URL each pair consists of a 4 byte (base64) string. These decode to 3 8-bit bytes. The last contains 4-bits from reading0 and 4-bits from reading 1.

When this class is instantiated, the 3 8-bit bytes are converted back into two 12-bit readings.

Parameters

- **rd0MSB** (*int*) – Most Signficant Byte of environmental sensor reading0.
- **rd1MSB** (*int*) – Most Significant Byte of environmental sensor reading1.
- **Lsb** (*int*) – Upper 4-bits are the least significant bits of reading0. Lower 4-bits are the least significant bits of reading1.

classmethod from_b64(*pairb64: str*)

Parameters **str4** (*str*) – A 4 character string that represents a base64 encoded pair. These are extracted from the circular buffer.

Returns

Return type A pair instantiated from the 4 character string.

classmethod from_bytes(*bytes: bytes*)

Parameters **bytes** (*bytes*) – The 3 bytes that make up a pair rd0MSB, rd1MSB and Lsb.

Returns

Return type A pair instantiated from the 3 byte input.

readings()

Returns

Return type A dictionary containing both 12-bit readings.

5.4 Status

class `wscodec.decoder.status.`**`Status`**(*statb64: str*)

Decode the status string.

Parameters **`statb64`** – Value of the URL parameter that holds status information (after base64 encoding).

`get_batvoltagemv`()

Returns Battery voltage converted to mV.

`get_batvoltageraw`()

Returns Battery voltage as an 8-bit value.

`get_resetcauseraw`()

Returns Reset cause as an 8-bit value.

CHAPTER

SIX

C ENCODER

6.1 Enc C Reference

Defines

`HDC2021_TEMPRH`
Last character of the URL version string if the URL contains both temperature and relative humidity measurands.

`HDC2021_TEMPONLY`
Last character of the URL version string if the URL contains only temperature measurands.

`ENDSTOP_BYTE`
Last character of the endstop. Must be URL safe according to RFC 1738.

`BATV_RESETCAUSE` (*BATV*, *RSTC*)
Macro for creating a 16-bit batv_resetcause value from 8-bit CODEC_FEAT_30 and CODEC_SPEC_16 values.

Enums

`enum pairbufstate_t`
Values:

`enumerator pairbuf_initial`

`enumerator pair0_both`
Write pair0

`enumerator pair0_reading1`
Overwrite reading1 of pair0

`enumerator pair1_both`
Write pair1

`enumerator pair1_reading1`
Overwrite reading1 of pair1

Functions

void **`fram_write_enable`** (void)
Enable writes to FRAM. Should be defined in the processor-specific cuplTag project.

void **`fram_write_disable`** (void)
Disable writes to FRAM. Should be defined in the processor-specific cuplTag project.

`static` bool **`one_reading_per_sample`** (void)

`static` void **`incr_loopcounter`** (void)
Update loop counter and battery voltage in the preamble status field.

Calls a function to measure battery voltage, increases loopcount and clears the battery reset field. These data are base64 encoded and written to EEPROM. ndef_writepreamble overwrites the bufferred circular buffer blocks so these must be read again after with demi_restore.

`static` void **`set_elapsed`** (unsigned int *minutes*)
Update the endmarker of the endstop with elapsed time in minutes.

Parameters **`minutes`** – Minutes elapsed since the previous sample.

`static` void **`set_pair`** (*pair_t* **pair*, int *rd0*, int *rd1*)
Write one pair.

Parameters

- **`pair`** – Pointer to the pair that will be modified.
- **`rd0`** – Reading 0 (12 bits).
- **`rd1`** – Reading 1 (12 bits).

`static` void **`set_rd1`** (*pair_t* **pair*, int *rd1*)
Overwrite reading1 in a pair This is used when the format stipulates one reading per pair (see CODEC_FEAT_42).

Parameters

- **`pair`** – Pointer to the pair that will be modified.
- **`rd1`** – Reading 1 (12 bits).

unsigned int **`enc_getbatv`** (void)
Get battery voltage from the status field. This is a value from 0-255. Step size increases exponentially. 255 corresponds to 1V5. 0 is infinity.

void **`enc_init`** (unsigned int *resetcause*, bool *err*, unsigned int *batv*)
Initialise the encoder state machine. Writes the.

Parameters

- **`resetcause`** – 16-bit status value.
- **`err`** – Sets an error condition where data will not be logged to the URL circular buffer.
- **`batv`** – Battery voltage from ADC (not in mv).

void **`enc_setelapsed`** (unsigned int *minutes*)
Update the base64 encoded endstop and write it to the tag.

Parameters **`minutes`** – Minutes elapsed since the previous sample.

int **`enc_pushsample`** (int *rd0*, int *rd1*)
Push a sample containing up to two readings onto the circular buffer.

Parameters

- **`rd0`** – First reading in the sample e.g. temperature.
- **`rd1`** – Second reading in the sample (optional) e.g. relative humidity.

Returns 1 if the cursor has moved from the end to the start and data are being overwritten. Otherwise 0.

Variables

nv_t **`nv`**
Externally defined parameters stored in non-volatile memory.

int **`overwriting`** = 0

pair_t **`pairbuf`**[2] = {0}
Stores two unencoded 3-byte pairs.

unsigned int **`npairs`** = 0
Number of base64 encoded pairs in the circular buffer, starting from the endstop and counting backwards.

pairbufstate_t **`state`** = pairbuf_initial
Pair buffer write state.

endstop_t **`endstop`** = {0}
The 16 byte end stop.

stat_t **`status`** = {0}
Structure to hold unencoded status data.

`struct stat_t`

Public Members

uint16_t **`loopcount`**
Number of times the last demi in the circular buffer endstop has wrapped from the end to the beginning.

uint16_t **`resetsalltime`**
2-byte status. Bits are set according to stat_bits.h

uint16_t **`batv_resetcause`**
Battery voltage in mV

`struct endstop_t`

Public Members

char **`hashnb64`**[12]
MD5 length field containing a base64 encoded *hashn_t*.

char **`markerb64`**[4]
End-stop marker comprised of base64 encoded minutes since the previous sample and *ENDSTOP_BYTE*

`struct endmarker_t`

Public Members

char **elapsedLSB**
    Minutes elapsed since previous sample (Least Significant Byte).

char **elapsedMSB**
    Minutes elapsed since previous sample (Most Signficant Byte).

6.2 PairHist C Reference

This maintains a circular buffer named *pairhistory*. It contains all pairs that are in the NDEF message circular buffer stored in the NFC-readable EEPROM. A crucial difference is that *pairhistory* is stored in RAM, so its contents can be accessed quickly.

This allows for a hash to be taken of the unencoded circular buffer pairs, each time this list changes. The decoder uses this to verify that it has decoded the circular buffer faithfully: It must output the same list of pairs, as fed to the encoder with multiple calls to *enc_pushsample()*.

After decoding the circular buffer, the hash is calculated. The decoder checks this equals the encoder hash, which is extracted from *endstop_t::hashnb64* in the NDEF message. If it does not, an error is raised and no data are returned.

The hash function can either be MD5 or HMAC-MD5 The former is a simple checksum for debugging the codec. It should not be used in production, because it is no good as a hash function and collisions can be found easily. The HMAC-MD5 should be used instead. A detailed discussion can be found in the CODEC_FEAT_24.

When a pair is pushed to or overwritten in the NDEF message, pairhistory must be updated with *pairhist_push()* and *pairhist_ovr()* respectively. This ensures that the output of *pairhist_hash()* will be accurate.

struct pair_t
    #include <pairhist.h> Structure to hold one sample consisting of two 12-bit readings.

Public Members

unsigned char **rd0Msb**
    Reading0 Most significant byte.

unsigned char **rd1Msb**
    Reading1 Most significant byte.

unsigned char **Lsb**
    Least significant 4-bit nibbles of reading0 and reading1.

struct hashn_t
    #include <pairhist.h> Structure to hold hash and npairs as per CODEC_SPEC_14.

Public Members

unsigned char **hash**[7]
    Last 7 bytes of the MD5 or HMAC-MD5 hash.

unsigned char **npairs**[2]
    Number of valid pairs in the circular buffer.

Defines

MD5DIGESTLEN_BYTES
Length of the MD5 digest (output) in bytes.

MD5BLKLEN_BYTES
Length of the MD5 input message block in bytes.

Functions

void **fram_write_enable**(void)
Enable writes to FRAM. Should be defined in the processor-specific cuplTag project.

void **fram_write_disable**(void)
Disable writes to FRAM. Should be defined in the processor-specific cuplTag project.

void **pairhist_ovr**(*pair_t pair*)
Overwrites the most recent pair in the history buffer. This is used when the format stipulates one reading per sample (rather than one pair per sample). For the first sample, a full pair is written with The second reading is set to an invalid value. On the next sample, the second reading in the pair is overwritten, so the history buffer must be overwritten with pairhist_ovr.

Parameters **pair** – New value of the most recent pair.

void **pairhist_init**()
Initialise the cursorindex The circular buffer itself (pairhistory) does not need to be initialised.

void **pairhist_push**(*pair_t pair*)
Pushes a new pair onto the history buffer. This operation overwrites an old pair if the circular buffer is full.

Parameters **pair** – Value of the new pair.

pair_t **pairhist_read**(unsigned int *offset*, int **error*)
Reads one pair at an offset from the end of pairhistory. This function makes it possible to read pairhistory as if it was a linear buffer.

Parameters

- **offset** – When 0, the most recent pair is returned. When 1, the 2nd most recent pair is returned. When BUFLEN_PAIRS-1, the oldest pair is returned. Any larger offset is invalid.
- **error** – Pointer to an error variable. This is set to 1 when offset exceeds the length of the circular buffer (BUFLEN_PAIRS-1). It is 0 otherwise.

Returns A pair read from pairhistory or a struct containing 3 zeroes if an error has occurred.

hashn_t **pairhist_hash**(int *npairs*, int *usehmac*, unsigned int *loopcount*, unsigned int *resetsalltime*, unsigned int *batv_resetcause*, int *endstopindex*)
Calculates a hash from *pairhistory* and other state variables.

The hash is calculated according to the table in CODEC_FEAT_24. If HMAC is enabled then the output is the last seven bytes of the HMAC-MD5 digest. If it is not then the hash is an MD5 checksum only. Note that the latter is intended for debug purposes only.

The MD5 is calculated iteratively 64 bytes at a time with multiple calls to MD5_Update().

Parameters

- **npairs** – The number of pairs from *pairhistory* to include in the hash.
- **usehmac** – When 0 the hash is MD5 only. Otherwise it is HMAC-MD5.

- **loopcount** – Number of times the circular buffer cursor has looped (or wrapped) from the end to the beginning.
- **resetsalltime** – Number of times the host firmware has logged a Power-on-Reset.
- **batv_resetcause** – 8-bit battery voltage concatenated with the 8-bit resetcause variable.
- **endstopindex** – Offset of the *ENDSTOP_BYTE* relative to the start of the NDEF message circular buffer.

Returns A value of type *hashn_t*. This contains the last 7 hash bytes together with npairs.

Variables

nv_t **nv**

pair_t **pairhistory**[**BUFLEN_PAIRS**] = {0}
Array of unencoded pairs. This mirrors the circular buffer of base64 encoded pairs stored in EEPROM.

int **cursorindex** = -1
Index marking the end of the circular buffer. The most recent sample is stored here. The next index contains the oldest sample.

unsigned char **msgblock**[**MD5BLKLEN_BYTES**]
Block to hold message data as an input to the MD5 algorithm.

const int **buflenpairs** = BUFLEN_PAIRS
Length of the circular buffer in pairs.

static const char **ipadchar** = 0x36
Inner padding byte for HMAC as defined in RFC 2104.

static const char **opadchar** = 0x5C
Outer padding byte for HMAC as defined in RFC 2104.

static MD5_CTX **ctx**
MD5 context.

6.3 Demi C Reference

Writes to a circular buffer of 8-byte demis. This is stored in an NFC readable EEPROM e.g. the NXP NT3H2111.

An EEPROM block is 16 bytes long. Demi is short for demi-block; it is 8 bytes long. A majority of transactions write 3 demis:

- Demi0: Two base64 encoded pairs (*pair_t*) comprised of 4x sensor readings.
- Demis1 and 2: Circular buffer endstop (*endstop_t*).

Demis are written to an EEPROM location given by *_cursordemi*:

- Even values of *_cursordemi* start at byte 0 of an EEPROM block.
- Odd values of *_cursordemi* start at byte 8 of an EEPROM block. A demi always fits completely into one EEPROM block, it never stradles two.

The function *demi_movecursor()* adds 1 to *_cursordemi* or resets it to 0 if the end of the circular buffer *_enddemi* has been reached.

There is no need to move the cursor after every write; the same 3 demis can be overwritten. If only one of the two available pairs in Demi0 changes at a time, the cursor is only moved after both have been produced. This applies if the format specifies OnePairPerSample (see CODEC_FEAT_42).

Sometimes only one demi needs to be overwritten: Demi2 contains minutes elapsed since the previous sample (::markerb64). This is overwritten every minute between samples. For this only one EEPROM block needs to be modified with *demi_commit2()*. This saves power, because writing to an I2C EEPROM is slow.

When all 3 demis are modified, 4 demis (two EEPROM blocks) must be written with *demi_commit4()*.

Whilst the code allows for 1,2 or 3 demis to be edited locally, the EEPROM must be read and written in multiples of 2 demis i.e. one block at a time. Two blocks of EEPROM are buffered at all times. This buffer must be updated:

- After the cursor is moved to a new location with *demi_movecursor()* or *demi_init()*.
- Before any write operations with *demi_write()*

This preserves data in the extra demi, which will either be after demi2 or before demi0. The buffer update is done with *demi_readcursor()*. It deduces which EEPROM blocks to copy into the local buffer based on *_cursordemi*.

Defines

DEMI0
Write first demi after the cursor.

DEMI1
Write second demi after cursor.

DEMI2
Write third demi after the cursor.

Typedefs

typedef enum *DemiState* **DemiState_t**
Structure to describe the state of the EEPROM circular buffer.

Enums

enum DemiState
Structure to describe the state of the EEPROM circular buffer.

Values:

enumerator ds_consecutive
_cursorblk is not 0 and *_nextblk* is located at the next EEPROM block.

enumerator ds_looparound
_cursorblk is at the end of the circular buffer and *_nextblk* is at the beginning. Data are overwritten from the first time this occurs.

enumerator ds_newloop
_cursorblk is 0 and *_nextblk* is 1. A new loop of the circular buffer has started.

Defines

`DEMI_TO_BLK`(*demi*)
: Maps a demi to its EEPROM block.

`IS_ODD`(*x*)
: Returns 1 if x is ODD and 0 if x is EVEN.

Functions

void **`fram_write_enable`**(void)
: Enable writes to FRAM. Should be defined in the processor-specific cuplTag project.

void **`fram_write_disable`**(void)
: Disable writes to FRAM. Should be defined in the processor-specific cuplTag project.

`static` void **`demi_read4`**(void)
: Copy 4 demis from EEPROM into RAM.

  This is 2 demis from *_cursorblk* and 2 demis from the block after it *_nextblk*. If *_cursorblk* is at the end of the buffer, then *_nextblk* will be at the start. This makes the buffer circular.

`static` void **`demi_shift2read2`**(void)
: Right shift the RAM buffer by 2 demis and append 2 demis read from the *_nextblk*.

  First: RAM buffer is right shifted by one block, overwriting the previous cursor block with the new cursor block. Second: New contents of *_nextblk* are copied out of EEPROM into the vacant RAM buffer block.

  The right shift saves a slow and unnecessary read of *_cursorblk* from EEPROM.

void **`demi_commit4`**(void)
: Copy 4 demis from RAM to EEPROM.

void **`demi_commit2`**(void)
: Write the last 2 demis from RAM to the EEPROM.

  Some functions only need to modify the last 2 demis so this saves time and energy over writing 4.

void **`demi_init`**(**`const`** int *startblk*, **`const`** int *lenblks*)
: Initialise the EEPROM circular buffer.

  Reads the first 4 demis in RAM.

  Parameters

  - **`startblk`** – EEPROM block to start the circular buffer.
  - **`lenblks`** – Length of circular buffer in EEPROM blocks.

int **`demi_write`**(int *offsetdemis*, char **demidata*)
: Overwrite one demi in the RAM buffer.

  The function to modify the RAM buffer *eep_cp()* requires a byte index relative to *_cursorblk*.

  - When *_cursordemi* is EVEN, nothing is needs to be done because it lies on a block boundary.
  - When *_cursordemi* is ODD then it is offset from the block boundary by one demi. Therefore one is added to offsetdemis.

  Parameters

  - **`offset`** – Demi index to overwrite, relative to _cursordemi. Must be 0, 1 or 2.

- **`demidata`** – Pointer to an 8 byte array of new demi data.

DemiState_t **`demi_movecursor`** (void)

void **`demi_readcursor`** (void)

Update RAM buffer to contain the 4 demis after _cursordemi.

This function must be called each time the cursor position is changed.

When *_cursordemi* is 0 it is assumed that the RAM buffer is empty, so all 4 demis are read. When *_cursordemi* is not 0, it is assumed that the RAM buffer has been populated before. It is also assumed that *_cursordemi* has only moved once since the previous time this function was called. Therefore it is not necessary to read 4 more demis out of the EEPROM.

int **`demi_getendmarkerpos`** (void)

Variables

int **`_endblk`** = 0

Last EEPROM block in the circular buffer.

int **`_startblk`** = 0

First EEPROM block in the circular buffer.

int **`_cursorblk`** = 0

Cursor address in terms of 16-byte EEPROM blocks. Must be >= *_startblk* and <= *_endblk*.

int **`_nextblk`** = 0

Address of the next EEPROM block after cursor block. The buffer is circular, so it can be < *_cursorblk*.

int **`_enddemi`** = 0

Largest possible value of _cursordemi. Always an odd integer.

int **`_cursordemi`** = 0

Cursor in terms of 8-byte demis. Must be >= 0 and <= *_enddemi*.

6.4 NVType C Reference

A file for organising configuration data stored in Non-Volatile memory.

These data are read by several parts of the encoder, where it is declared as an external global variable.

The variable definition depends on how the encoder is being run:

- When running under CFFI (see PyEncoder) *nv* is defined in nvtype.c
- When running as part of a larger project (e.g. the cupl Tag firmware) nv must be defined elsewhere.

The intention is for nv to occupy the 512 byte MSP430 information FRAM.

Date 6 Aug 2018

Author Malcolm Mackay

Copyright Plotsensor Ltd.

Defines

`SERIAL_LENBYTES`
Length of the tag serial string in bytes.

`SECKEY_LENBYTES`
Length of the secret key used for HMAC-MD5 in bytes.

`BASEURL_LENBYTES`
Maximum length of the base URL string in bytes.

`SMPLINT_LENBYTES`
Length of the sample interval (minutes) integer in bytes.

`VFMTINT_LENBYTES`
VFmt character array length in bytes.

`FORMAT_ASCII_MAXLEN`
Maximum length of the format ASCII string.

`MINVOLTAGEMV_ASCII_MAXLEN`
Maximum length of the minimum voltage (mV) ASCII string.

`SMPLINTERVAL_ASCII_MAXLEN`
Maximum length of the sample interval string.

Typedefs

`typedef struct` *nvstruct* **`nv_t`**
Structure to hold configuration data held in non-volatile memory.

`struct nvstruct`
#include <nvtype.h> Structure to hold configuration data held in non-volatile memory.

Public Members

char **`serial`**[**`SERIAL_LENBYTES`**]
Alphanumeric serial of the tag running the cupl encoder.

char **`seckey`**[**`SECKEY_LENBYTES`**]
Secret key string used for HMAC-MD5.

char **`smplintervalmins`**[**`SMPLINT_LENBYTES`**]
Time interval betweeen samples in minutes.

char **`baseurl`**[**`BASEURL_LENBYTES`**]
URL of the cupl Web Application frontend.

char **`format`**
Codec format byte.

unsigned int **`minvoltagemv`**
Minimum startup voltage in mV.

unsigned int **`usehmac`**
When non-zero enable HMAC otherwise use MD5 only.

unsigned int **`httpsdisable`**
When non-zero use HTTP in the URL otherwise use HTTPS.

unsigned int **`sleepintervaldays`**
  Number of days to wait without scans before putting the cupl Tag into deep sleep mode.

unsigned int **`allwritten`**
  When non-zero all required NV parameters have been set.

unsigned int **`resetsperloop`**
  Incremented each time the tag microcontroller resets. Zeroed when the circular buffer loops around (see *ds_looparound*).

unsigned int **`resetsalltime`**
  Incremented each time the tag microcontroller resets.

Functions

void **`fram_write_enable`**()
  Enable writes to FRAM. Should be defined in the processor-specific cuplTag project.

void **`fram_write_disable`**()
  Disable writes to FRAM. Should be defined in the processor-specific cuplTag project.

Variables

`nv_t nv = {.serial="AAAACCCC", .seckey="AAAACCCC"}`
  Externally defined parameters stored in non-volatile memory.

6.5 Eep C Reference

Defines

`BUFSIZE_BLKS`

`BUFSIZE_BYTES`

Functions

void **`fram_write_enable`**(void)
  Enable writes to FRAM. Should be defined in the processor-specific cuplTag project.

void **`fram_write_disable`**(void)
  Disable writes to FRAM. Should be defined in the processor-specific cuplTag project.

`static inline` int **`inbounds`**(int *byteindex*)
  Checks if a byte index is within the bounds the buffer array.

  This can be used to prevent the program from accessing memory that is out of bounds.

  Parameters **`byteindex`** – Index of a buffer byte that is to be read or written.

  Returns 0 if the index is less than the size of the buffer and can be accessed safely. Otherwise 1.

int **`eep_write`**(**`const`** int *eepblk*, **`const`** unsigned int *bufblk*)
  Write a 16-byte block from the buffer to EEPROM.

  Parameters

- **`eepblk`** – Block of the EEPROM to write to.
- **`bufblk`** – Block of the buffer to write from.

Returns 1 if the block to be written greater than the buffer size. 0 on success and -1 on write error.

void **`eep_waitwritedone`**()
Block until the EEPROM block write has finished.

Writes of Flash memory take some milliseconds to complete.

int **`eep_read`**(**`const`** int *eepblk*, **`const`** unsigned int *bufblk*)
Read a 16-byte block from EEPROM to the buffer.

Parameters

- **`eepblk`** – EEPROM block to read.
- **`bufblk`** – Block of the buffer to copy the EEPROM contents to.

int **`eep_swap`**(**`const`** unsigned int *srcblk*, **`const`** unsigned int *destblk*)
Swap two buffer blocks.

Parameters

- **`srcblk`** – The buffer block to read from.
- **`destblk`** – The buffer block to write to.

int **`eep_cp`**(int **indexptr*, **`const`** char **dataptr*, **`const`** int *lenbytes*)
Copy data from a pointer into the buffer.

Parameters

- **`indexptr`** – Data are copied into the buffer starting from this index. An integer from 0 to N-1, where N is the size of the buffer. indexptr is overwritten by the index one greater than the last data to be written.
- **`dataptr`** – Data are copied from this pointer.
- **`lenbytes`** – The number of bytes to copy into the buffer from dataptr.

Returns 0 if the data to be copied will fit entirely in the buffer. Otherwise 1.

int **`eep_cpbyte`**(int **indexptr*, **`const`** char *bytedata*)
Copy one byte into the buffer.

Parameters

- **`indexptr`** – The byte is copied into this index of the buffer. An integer from 0 to N-1, where N is the size of the buffer. indexptr is overwritten by indexptr+1.
- **`bytedata`** – Byte to be copied into the buffer.

Returns 0 if indexptr is an index that will not overflow the buffer. Otherwise 1.

Variables

char **`_blkbuffer`**[**`BUFSIZE_BLKS`** * **`BLKSIZE`**] = {0}

6.6 NDEF C Reference

Defines

`URL_RECORDTYPE`
NDEF record type for a URL.

`URL_RECORDTYPE_LEN`
Length of the NDEF record type in bytes.

`SMPLINTKEY_LEN`
Length of the sample interval key string in bytes.

`SMPLINTB64_LEN`
Length of the encoded sample interval string in bytes.

`SERIALKEY_LEN`
Length of the serial key string in bytes.

`VERKEY_LEN`
Length of the vfmt key string in bytes.

`VFMTB64_LEN`
Length of the encoded VFmt data in bytes.

`STATKEY_LEN`
Length of the status key string in bytes.

`STATB64_LEN`
Length of the encoded *status* string in bytes.

`CBUFKEY_LEN`
Length of the circular buffer key string in bytes.

`NDEF_RECORD_HEADER_LEN`
Length of the NDEF record header in bytes.

`TL_LEN`
Length of the Tag and Length fields of the NDEF message TLV in bytes.

`TAG_NDEF_MESSSAGE`
Tag indicating the TLV block contains an NDEF message.

`WELLKNOWN_TNF`
Record Type follows the Record Type Definition (RTD) format.

`URI_ID_HTTP`
URI Identifier Code for the HTTP protocol.

`URI_ID_HTTPS`
URI Identifier Code for the HTTPS protocol.

Functions

`static` void **`ndef_createurlrecord`** (int **eepindex*, int *msglenbytes*, int *httpsDisable*)
Create a URL NDEF Record.

Parameters

- **`eepindex`** – Position in the 64-byte array that buffers data to be written into EEPROM.
- **`msglenbytes`** – NDEF Message Length in bytes.

void **`ndef_calclen`** (int **paddinglen*, int **preamblenbytes*, int **urllen*)

int **`ndef_writepreamble`** (int *buflenblks*, char **statusb64*)
Write the part of the URL before the circular buffer.

Parameters

- **`buflenblks`** – Circular buffer length in 16-byte EEPROM blocks.
- **`statusb64`** – Pointer to a base64 encoded *status* structure.

Returns 1 if buflenblks is not even.

void **`ndef_writeblankurl`** (int *buflenblks*, char **statusb64*, int **bufstartblk*)
Write an NDEF message containing one URL record to EEPROM. The URL contains a circular buffer. This is populated with a placeholder text - all zeroes - initially.

Parameters

- **`buflenblks`** – Circular buffer length in 16-byte EEPROM blocks.
- **`statusb64`** – Pointer to a base64 encoded *status* structure.
- **`bufstartblk`** – The circular buffer start block is written to this pointer.

Variables

nv_t **`nv`**

`static const` char **`serialkey`**[] = "&s="
Seperator, key and equals before the serial string.

`static const` char **`cbufkey`**[] = "&q="
Seperator, key and equals before the circular buffer string.

`static const` char **`verkey`**[] = "&v="
Seperator, key and equals before the vfmt string.

`static const` char **`statkey`**[] = "&x="
Seperator, key and equals before the status string.

`static const` char **`smplintkey`**[] = "/?t="
Start of parameters followed by a key and equals for the sample interval string.

`static const` char **`zeropad`**[] = "MDAw"
4 characters that base64 decode to 0,0,0

`union TNFFlags_t`

Public Members

unsigned char **all**

unsigned char **tnf**
Type Name Format field.

unsigned char **idpresent**
ID present flag. 1 if the ID field is present.

unsigned char **srecord**
Short record flag. 1 if the payload length field is 1 byte long.

unsigned char **chunkflag**
Chunk flag. 1 if this is the first or middle record in a chunked message.

unsigned char **msgend**
Message end flag. 1 if this is the last record in the message.

unsigned char **msgbegin**
Message begin flag. 1 if this is the first record in the message.

struct *TNFFlags_t*::**[anonymous] byte**

union len_t

Public Members

unsigned long **all**

unsigned char **bytes**[4]

6.7 Defs Header File

Defines

CODEC_VERSION

BYTES_PER_DEMI
The number of bytes per demi.

DEMIS_PER_BLK
The number of demis per block.

PAIRS_PER_DEMI
The number of base64 encoded pairs per demi.

BUFLEN_BLKS
Length of the circular buffer in 16-byte blocks.

ENDSTOP_BLKS
Endstop length in 16-byte blocks.

ENDMARKER_OFFSET_IN_ENDSTOP_1

BLKSIZE

BUFLEN_PAIRS

6.8 NT3H C Reference

Warning: doxygenfile: Cannot find file "nt3h.c

CHAPTER

SEVEN

PYTHON WRAPPED ENCODER (PYENCODER)

7.1 Mock EEPROM

class `wscodec.encoder.pyencoder.eeprom.`**`Eeprom`**(*sizeblocks: int*)

A mock of the NT3H2111 EEPROM, based on a bytearray. There are methods to read and write from this in 16-byte blocks. Helper methods parse the entire EEPROM contents as an NDEF message. This mimics what a phone will do when it reads the NT3H2111 using NFC.

The EEPROM will normally contain the output of the cupl Encoder: an NDEF message containing one NDEF record. This itself will contain a URL. One of the parameters in the query string is 'q', which contains the circular buffer of temperature and relative humidity samples.

`decode_ndef`() → ndef.record.Record

Decode the NDEF message.

Returns First NDEF record in the message

`display_block`(*block: int*) → bytearray

Display one EEPROM block.

Parameters **`block`** – Address of the block to display.

Returns Block data as a list of 16 bytes.

`get_message`() → bytearray

Extract the NDEF message bytes from EEPROM.

Returns NDEF message bytearray

`get_qdemis`() → list

Returns The value of URL parameter 'q' as a list of 8-byte demis.

`get_qparam`() → str

Returns The value of URL parameter 'q' as a string.

`get_url`() → str

Obtain URL from the NDEF record stored in EEPROM.

Returns URL string

`get_url_parsed`() → urllib.parse.ParseResult

Parse URL in the EEPROM NDEF record.

Returns Parsed URL

`get_url_parsedqs`() → dict

Parse parameters in the query string

Returns A dictionary of URL parameters.

read_block(*block: int*, *blkdata: ctypes.c_char_p*)
Read one block into a pointer.

Parameters

- **block** – Address if the block to read.
- **blkdata** – Pointer to an array of 16 bytes that the block will be read into.

Returns None

write_block(*block: int*, *blkdata: ctypes.c_char_p*)
Write one block from a pointer.

Parameters

- **block** – Address of the block to write
- **blkdata** – Pointer to an array of 16 bytes that will be written to the block.

Returns None

class wscodec.encoder.pyencoder.instrumented.**InstrumentedBase**(*ffimodule*, *baseurl*, *serial*, *secretkey*, *smplintervalmins*, *batteryadc=100*, *format=1*, *resetsalltime=0*, *usehmac=True*, *httpsdisable=False*)

Put some documentation here

class wscodec.encoder.pyencoder.instrumented.**InstrumentedDemi**(*baseurl='plotsensor.com'*, *serial='AAAACCCC'*, *secretkey='AAAACCCC'*, *smplintervalmins=12*)

class wscodec.encoder.pyencoder.instrumented.**InstrumentedNDEF**(*baseurl='plotsensor.com'*, *serial='AAAACCCC'*, *secretkey='AAAACCCC'*, *smplintervalmins=12*)

class wscodec.encoder.pyencoder.instrumented.**InstrumentedPairhist**(*baseurl='plotsensor.com'*, *serial='AAAACCCC'*, *secretkey='AAAACCCC'*, *smplintervalmins=12*)

class wscodec.encoder.pyencoder.instrumented.**InstrumentedSample**(*baseurl='plotsensor.com', serial='AAAACCCC', secretkey='AAAACCCC', smplintervalmins=12, batteryadc=100, format=1, resetsalltime=0, usehmac=True, httpsdisable=False*)

pushsamplelist(*trhlist: list*)

Parameters **trhlist** – a list of dictionaries each containing temperature and relative humidity keys.

Returns None

rh_percent_to_raw(*rhpc*)

Converts from relative humidity in percent to a raw ADC value for the Texas HDC2010.

temp_degc_to_raw(*degc*)

Converts degrees C to a raw ADC value for the Texas HDC2010.

updateendstop(*minutes: int*)

Update the endstop with minutes elapsed since the most recent sample.

Parameters **minutes** – Minutes elapsed since the most recent sample.

Returns None

class wscodec.encoder.pyencoder.instrumented.**InstrumentedSampleT**(*serial='ABCDEFGH', secretkey='AAAACCCC11112222', baseurl='plotsensor.com', smplintervalmins=12, resetsalltime=0, batteryadc=100, resetcause=0, usehmac=True, httpsdisable=False, tagerror=False, format=2*)

class wscodec.encoder.pyencoder.instrumented.**InstrumentedSampleTRH**(*serial='ABCDEFGH'*, *secretkey='AAAACCCC11112222'*, *baseurl='plotsensor.com'*, *smplintervalmins=12*, *resetsalltime=0*, *batteryadc=100*, *resetcause=0*, *usehmac=True*, *httpsdisable=False*, *tagerror=False*, *format=1*)

class wscodec.encoder.pyencoder.unitc.**CFFIGenerator**(*blacklist*)

class wscodec.encoder.pyencoder.unitc.**FunctionList**(*source*)

wscodec.encoder.pyencoder.unitc.**load**(*filename*, *depfilenames=[]*)

Load a file

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX