cupl Tag Documentation

Release 0.0.1

Malcolm Mackay

Aug 26, 2022

FIRMWARE

CHAPTER

ONE

STATE CHART

This might look intimidating at first. Sub-sections cover the most common progressions through the state chart.

1.1 Not Configured

1.2 Programming Mode

1.3 First Run

The following state chart shows how the cuplTag operates the first time it is powered on with a good battery and a complete configuration.

After some initialisation, a sample is collected from the HDC2021 and fed into cuplcodec.

1.4 Sampling Loop

The cuplTag wakes up each minute. Several initialisation states are skipped. The minute counter is incremented. If this equals the configured sampling interval, then a sample is collected from the HDC2021.

cuplcodec updates the minute counter in the cupl URL (stored on the NFC EEPROM). If a new sample is available, this is encoded and added to the circular buffer.

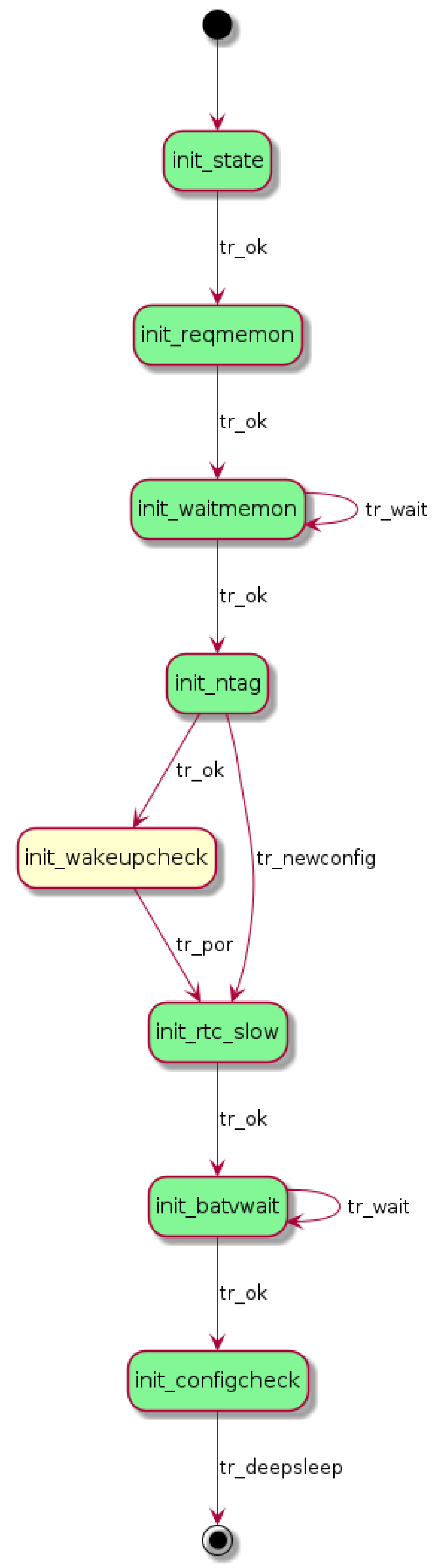

Fig. 1: Startup does not continue when the configuration is incomplete.

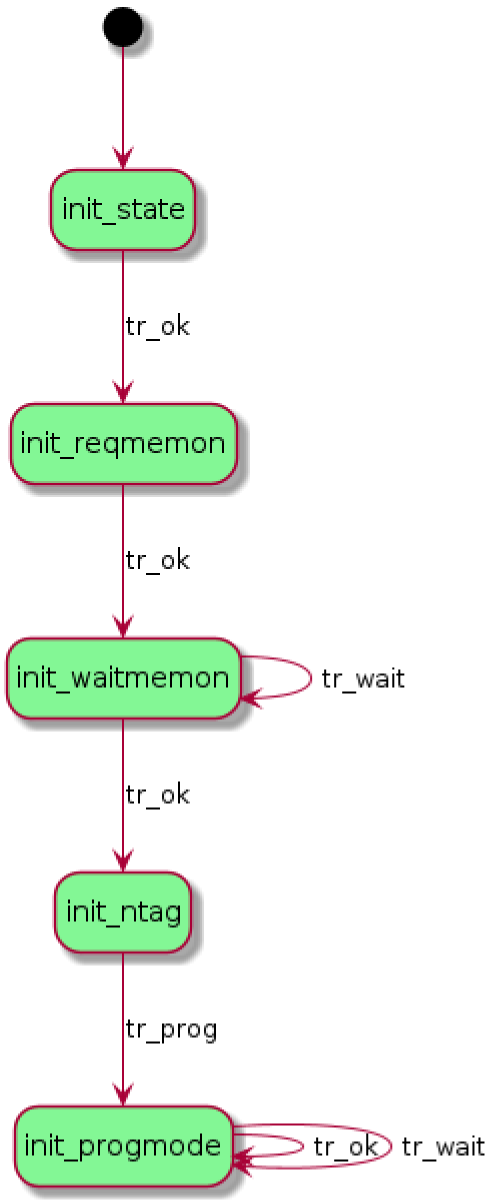

Fig. 2: The state machine branches to `init_progmode()` when the nPRG pin is LOW.

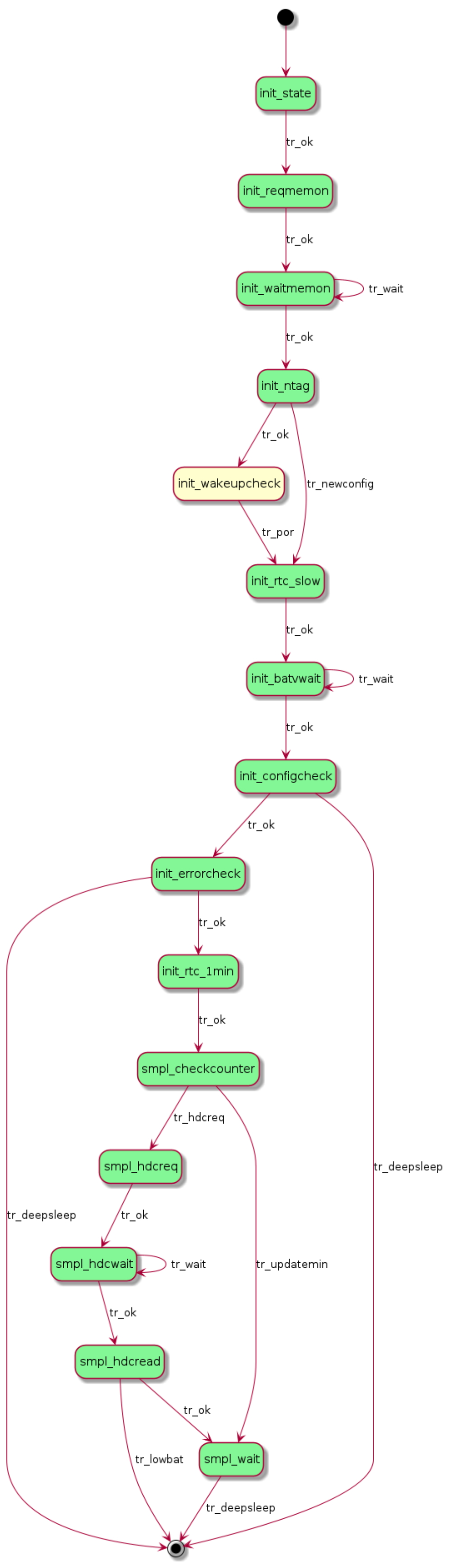

Fig. 3: The state machine in `main()`.

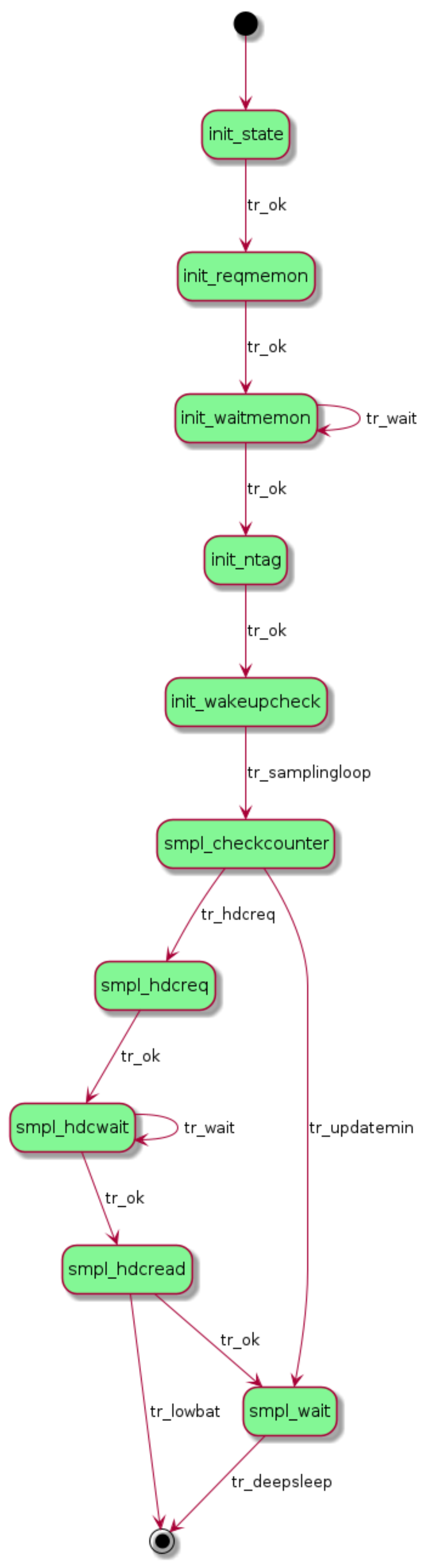

Fig. 4: The state machine runs each minute in the sampling loop.

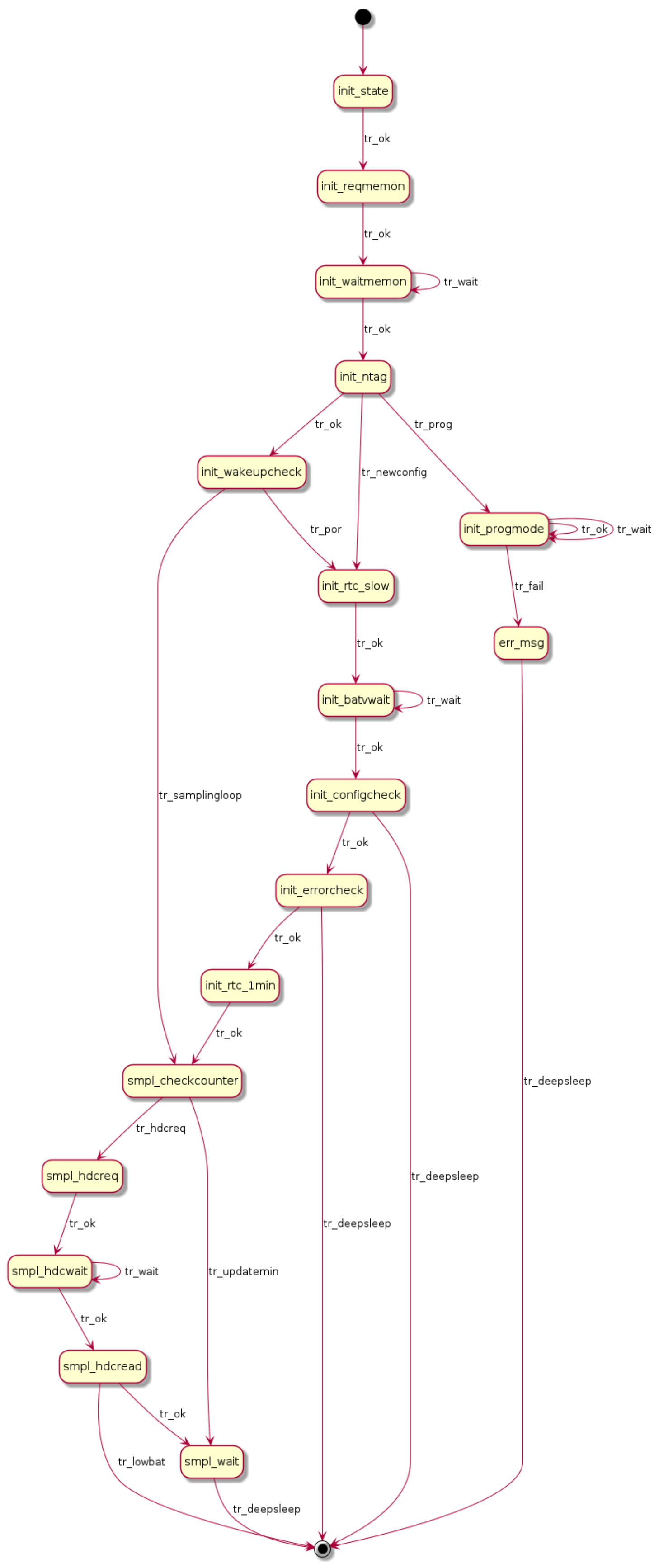

Fig. 5: The state machine in `main()`.

CHAPTER

TWO

REFERENCE

2.1 Main

Top-level Finite State Machine for controlling the MSP430 and cuplTag as a whole.

A Finite State Machine is defined in this file and run from *main()*. This has several features.

Author

Malcolm Mackay

It makes calls to drivers for communicating with the HDC2022 humidity sensor and the NT3H2111 NFC EEPROM.

It controls entry into the 'programming mode' sub state machine, where configuration strings can be written using a serial port.

It reads configuration strings from the NFC EEPROM if any are present.

It collects samples from an HDC2022 at a fixed time interval and passes these to the cuplcodec encoder.

Defines

`CS_SMCLK_DESIRED_FREQUENCY_IN_KHZ`

Target frequency for SMCLK in kHz.

`CS_XT1_CRYSTAL_FREQUENCY`

Resonant frequency of the XT1 crystal in kHz.

`CS_XT1_TIMEOUT`

Timeout for XT1 to stabilise at the resonant frequency in SMCLK cycles.

`CP10MS`

ACLK Cycles Per 10 MilliSeconds. Assumes ACLK = 32768 kHz and a divide-by-8.

`EXIT_STATE`

State machine exit state.

`ENTRY_STATE`

State machine entry state.

Typedefs

typedef enum *state_codes* **`tstate`**

States in the Finite State Machine are represented by a code. This is used each time the state machine is run, to determine:

a. Which state function to call in *main()*.
b. The next state in *lookup_transitions()*.

typedef enum *ret_codes* **`tretcode`**

Each state function returns at least one code from the list below. This is used by *lookup_transitions()* to determine the next state.

typedef enum *event_codes* **`tevent`**

Events occur asynchronously to execution of the FSM. The MSP430 will typically wait for an event in sleep mode to save power. An example is an edge on the INT (interrupt) output from a temperature sensor. This is connected to an input on the MSP430, which is configured to call an Interrupt Service Routine (ISR). The ISR sets a flag and 'wakes up' the MSP430 from sleep mode.

The *main()* function is entered, flags are checked then cleared and the event variable is set according to the list of codes below. Finally, the state function is called with the event passed as an argument.

Multiple events can occur simultaneously, but only one at-a-time is passed to the FSM.

Enums

enum **`state_codes`**

States in the Finite State Machine are represented by a code. This is used each time the state machine is run, to determine:

a. Which state function to call in *main()*.
b. The next state in *lookup_transitions()*.

Values:

enumerator **`sc_init`**

State code for *init_state()*

enumerator **`sc_init_reqmemon`**

State code for *init_reqmemon()*

enumerator **`sc_init_waitmemon`**

State code for *init_waitmemon()*

enumerator **`sc_init_ntag`**

State code for *init_ntag()*

enumerator **`sc_init_progmode`**

State code for *init_progmode()*

enumerator **`sc_init_configcheck`**

State code for *init_configcheck()*

enumerator **`sc_init_errorcheck`**

State code for *init_errorcheck()*

enumerator **`sc_init_wakeupcheck`**

State code for *init_wakeupcheck()*

enumerator **`sc_init_batvwait`**

State code for *init_batvwait()*

enumerator **`sc_init_rtc_slow`**

State code for *init_rtc_slow()*

enumerator **`sc_init_rtc_1min`**

State code for *init_rtc_1min()*

enumerator **`sc_smpl_checkcounter`**

State code for *smpl_checkcounter()*

enumerator **`sc_smpl_hdcreq`**

State code for *smpl_hdcreq()*

enumerator **`sc_smpl_hdcwait`**

State code for *smpl_hdcwait()*

enumerator **`sc_smpl_hdcread`**

State code for *smpl_hdcread()*

enumerator **`sc_smpl_wait`**

State code for *smpl_wait()*

enumerator **`sc_err_msg`**

State code for *err_msg()*

enumerator **`sc_end`**

State code for *end_state()*

enum **`ret_codes`**

Each state function returns at least one code from the list below. This is used by *lookup_transitions()* to determine the next state.

Values:

enumerator **`tr_ok`**

enumerator **`tr_prog`**

enumerator **`tr_newconfig`**

enumerator **`tr_hdcreq`**

Request a sample from the HDC2021 sensor.

enumerator **`tr_updatemin`**

enumerator **`tr_deepsleep`**

cuplTag should enter a deep sleep state LPM3.5

enumerator **`tr_lowbat`**

enumerator **`tr_fail`**

enumerator **`tr_samplingloop`**

cuplTag is in the sampling loop. The reset was caused by an exit from LPM3.5

enumerator **`tr_por`**

cuplTag is NOT in the sampling loop. A Power-On-Reset has occurred.

enumerator **`tr_wait`**

cuplTag should enter a sleep state, such as LPM0, to wait for an event.

enum **`event_codes`**

Events occur asynchronously to execution of the FSM. The MSP430 will typically wait for an event in sleep mode to save power. An example is an edge on the INT (interrupt) output from a temperature sensor. This is connected to an input on the MSP430, which is configured to call an Interrupt Service Routine (ISR). The ISR sets a flag and 'wakes up' the MSP430 from sleep mode.

The *main()* function is entered, flags are checked then cleared and the event variable is set according to the list of codes below. Finally, the state function is called with the event passed as an argument.

Multiple events can occur simultaneously, but only one at-a-time is passed to the FSM.

Values:

enumerator **`evt_none`**

No event has occurred.

enumerator **`evt_timerfinished`**

The timer peripheral has counted down to 0.

enumerator **`evt_hdcint`**

Pin change interrupt received from the HDC2021 temperature and humidity sensor.

Functions

void **fram_write_enable**()

Enable writes to program FRAM.

Some variables are stored in program FRAM. RAM cannot be used because state is lost in deep sleep mode (LPM3.5). The Program FRAM Write Protect bit must be cleared (and interrupts disabled) before a write.

void **fram_write_disable**()

Disable writes to program FRAM.

Sets the Program FRAM Write Protect bit and re-enables interrupts.

tretcode **init_state**(*tevent* evt)

Initialise clocks and IOs on the MSP430.

All IOs are configured into an initial state. The number of IOs left as inputs (default) must be minimised to reduce power consumption.

The slow Auxiliary Clock (ACLK) is sourced form the external 32.768 kHz crystal. An internal 10 kHz source is used by default. This is power hungry and drifts with temperature.

Next, the Phased-Locked Loop (DCO) generates an output frequency of 1 MHz, by multiplying the external 32.768 kHz crystal frequency up by 31.

Internal clocks MCLK and SMCLK are connected to the DCO output.

1 MHz was selected to minimise current draw from the high impedance coin cell battery. This results in a lower voltage drop after exiting the sleep state. Battery life is limited by this voltage drop. This is not the case if the source impedance is lower. Then it is best to operate at a higher frequency: up to 24 MHz.

Finally, the cause of the reset is read. The program needs to know whether this is just a routine wake-up from sleep (LPM3.5) or the result of a fault.

tretcode **init_reqmemon**(*tevent* evt)

This state calls *reqmemon()*.

tretcode **init_waitmemon**(*tevent* evt)

This state calls *waitmemon()*.

tretcode **init_ntag**(*tevent* evt)

Initialise the dual-interface I2C+NFC EEPROM.

A call is made to '*nt3h_check_address()*' to make sure the EEPROM is at device address 0x55.

The first EEPROM block is read to check for an NFC text record. If found, configuration strings are extracted and saved into non-volatile memory.

The capability container is written if it needs to be. These 4 bytes indicate that the tag contains an NDEF message.

The programming mode select pin (nPRG) is checked. If it is LOW, the return code is updated.

tretcode **init_progmode**(*tevent* evt)

Run the programming mode sub-state machine.

Enables the serial port (UART) and responds to text commands.

It is only intended that this state be entered in a production environment, not by the end user.

The only way to exit is to reset the microcontroller. This can be done by sending a soft reset command '<z>'.

tretcode `init_configcheck`(*tevent* evt)

Verifies that all configuration strings have been written.

The state machine cannot continue unless the tag is fully configured. There are no default settings.

When configuration is incomplete, a text-based error message is written to the tag and deep-sleep mode is entered.

tretcode `init_errorcheck`(*tevent* evt)

Check for an error condition before continuing startup.

An error occurs if the reset was caused by a reason other than a new battery insertion.

When the battery voltage is below a threshold (set in NVM). The state machine should not get stuck in a loop, where an attempt is made to write to the NFC EEPROM before the micro-controller resets. Reset loops can wear out the EEPROM.

In the event of 10 consecutive resets that result in an error, or a single low battery reading:

a. Report the most recent error in the cupl URL status word.
b. Do not include any samples in the cupl URL.
c. Go to a deep sleep state that prevents another reset cycle from occurring for some time.

Otherwise:

a. Report the most recent error in the cupl URL but treat it as spurious.
b. Allow the state machine to continue.

Returns
tr_deepsleep when 10 consecutive errors have occurred or the battery voltage is low. Otherwise indicate no errors with *tr_ok*.

tretcode `init_wakeupcheck`(*tevent* evt)

Has the reset has been caused by a routine RTC wake-up?

In the sampling loop, Wake-ups from LPM3.5 occur every minute. These are invoked by an interrupt from the Real Time Clock peripheral.

This function first makes a call to *stat_rstcause_is_lpm5wu()*. Then it checks if the first integer in Backup Memory is 1. If it is, then the cuplTag is in the sampling loop.

Returns
tr_samplingloop if the cuplTag is in the sampling loop. Otherwise, indicate a power-on-reset with *tr_por*.

tretcode `init_batvwait`(*tevent* evt)

Wait for the battery voltage to stabilise.

It takes some time for capacitors to charge after a battery is inserted. A timer is started in the previous state. The MSP430 waits here in low power mode.

Parameters
`evt` – **[in]** Event. When set to *evt_timerfinished* the state machine progresses.

Returns
tr_ok when the timer has finished. Otherwise *tr_wait*.

tretcode **`init_rtc_slow`**(*tevent* evt)

Configure the Real Time Clock to peripheral to generate one interrupt every 30 minutes.

This is done to prevent the cuplTag from being stuck in the end_state when an error occurs during startup. This state must be entered before any possible transitions to the end_state.

Whatever the error, the cuplTag must wake up and try to start up again. A long time interval has been chosen so that the battery is not depleted.

tretcode **`init_rtc_1min`**(*tevent* evt)

Configure the Real Time Clock peripheral to generate one interrupt every minute.

The cuplTag spends most of the time in a deep sleep mode LPM3.5 to save power. Each minute, it wakes up for a few milliseconds to collect a sample or to increment *minutecounter*.

TURBO MODE is a special feature that is useful for test purposes. When enabled, the interrupt occurs every 3 seconds. To enable, set the time interval parameter to 0.

tretcode **`smpl_checkcounter`**(*tevent*)

tretcode **`smpl_hdcreq`**(*tevent* evt)

Request a sample from the HDC2021 sensor.

A sample is requested by the MSP430 with hdc2020_startconv(). When data is ready, the HDC2021 makes a HIGH to LOW transition on its INT pin. To save power, the MSP430 sleeps whilst the measurement is taken.

The MSP430 pin connected to INT is made to raise an interrupt when a falling edge is detected.

tretcode **`smpl_hdcwait`**(*tevent* evt)

Wait in LPM3 whilst waiting for a data ready interrupt from the HDC2021 sensor.

Returns

tr_ok when a DRDY interrupt has been detected. Otherwise return *tr_wait* to indicate that the MSP430 should sleep.

tretcode **`smpl_hdcread`**(*tevent*)

tretcode **`smpl_wait`**(*tevent*)

tretcode **`err_msg`**(*tevent* evt)

Write the NDEF message *ndefmsg_badtrns* to the NFC EEPROM.

Notify the developer that an invalid state machine transition has been requested. The user should never see this. This will occur if a transition has been requested that does not exist in the *state_transitions* table.

tretcode **`end_state`**(*tevent* evt)

Put the MSP430 into deep sleep LPM3.5.

Minimise power consumption by powering down as much of the MSP430 (and cuplTag) as possible. Only the Real Time Clock and Backup Memory peripherals remain powered on.

The RTC can generate an interrupt to wake the MSP430 up. When this occurs, the program starts from a reset condition. Only the backup memory can be used to persist state.

This function disables GPIO interrupt sources. It stops any timers, in case these prevent LPM3.5 from being entered. It disables the watchdog, because this is power hungry and the RTC can be used instead (see *init_rtc_slow()*).

Most importantly it calls *memoff()* to make sure that the VMEM domain is powered down.

The Supply Voltage Supervisor is disabled to save power. It is not very useful in deep sleep mode. The battery voltage will decline very little between wake-ups from the RTC.

Returns
tr_ok but this is to keep the compiler happy. Deep sleep is entered before the function returns.

tstate **lookup_transitions**(*tstate* curstate, *tretcode* rc)

Look up the next state in the Finite State Machine.

The look-up is performed by iterating through an array of transitions. An error state is returned if no match is found.

Parameters

- **curstate** – **[in]** Current state.
- **rc** – **[in]** Code returned from the current state.

Returns
Next state.

static void **writetxt**(const char *msgptr, int len)

Write an NDEF message to the NFC EEPROM.

The NDEF message normally contains one text record. It can be created with an external program and stored as a constant array. The function is used to display simple error messages to the end-user.

Parameters

- **msgptr** – **[in]** Pointer an NDEF message array.
- **len** – **[in]** Length of the NDEF message.

static void **wdog_kick**()

Kick the watchdog, to prevent it from timing out.

This is done by writing to the watchdog control register.

static void **start_timer**(unsigned int intervalCycles)

Start a single-shot timer.

An interrupt fires when the timer has finished counting. The MSP430 can sleep in LPM3 whilst waiting for it. This saves power over delay loops.

The function is best suited to pausing execution for a short time (milliseconds).

Parameters
intervalCycles – **[in]** Number of 4.096 kHz clock cycles to count.

static void **memoff**()

Power down the VMEM domain.

The load switch enable pin is set low, breaking the circuit between VDD and VMEM. This is done to save power in sleep mode. The NT3H2111 EEPROM will otherwise draw ~10uA.

static *tretcode* **reqmemon**(*tevent* evt)

Enable power to the VMEM domain.

Configure a pin to receive interrupts from the humidity sensor. Set the load switch enable pin HIGH to power up the VMEM domain from VDD.

After this function has been called, the MSP430 must sleep whilst waiting for a Timer interrupt. When this fires, the VMEM voltage should be stable.

static *tretcode* **waitmemon**(*tevent* evt)

Wait for the VMEM voltage to stabilise after power on.

Timer_B1 must be started with *start_timer()* prior to calling this function.

Parameters

evt – **[in]** Event. When set to evt_timerfinished, I2C is enabled and the state machine progresses.

void **main**(void)

void **TIMER1_B0_ISR**(void)

if ((P1IN &BIT1)==0)

Variables

int **timerFlag** = 0

Flag set by the Timer Interrupt Service Routine.

int **hdcFlag** = 0

Flag set by the HDC2021 humidity sensor data-ready Interrupt Service Routine.

int **minutecounter** = 0

Incremented each time the sampling loop is run.

const char **ndefmsg_progmode**[] = {0x03, 0x3D, 0xD1, 0x01, 0x39, 0x54, 0x02, 0x65, 0x6E, 0x50, 0x72, 0x6F, 0x67, 0x72, 0x61, 0x6D, 0x6D, 0x69, 0x6E, 0x67, 0x20, 0x4D, 0x6F, 0x64, 0x65, 0x2E, 0x20, 0x43, 0x6F, 0x6E, 0x6E, 0x65, 0x63, 0x74, 0x20, 0x74, 0x6F, 0x20, 0x73, 0x65, 0x72, 0x69, 0x61, 0x6C, 0x20, 0x70, 0x6F, 0x72, 0x74, 0x20, 0x61, 0x74, 0x20, 0x39, 0x36, 0x30, 0x30, 0x20, 0x62, 0x61, 0x75, 0x64, 0x2E, 0xFE}

Hard-coded NDEF message containing one text record "Programming Mode. Connect to serial port at 9600 baud."

const char **ndefmsg_noconfig**[] = {0x03, 0x2D, 0xD1, 0x01, 0x29, 0x54, 0x02, 0x65, 0x6E, 0x43, 0x6F, 0x6E, 0x66, 0x69, 0x67, 0x20, 0x63, 0x68, 0x65, 0x63, 0x6B, 0x20, 0x66, 0x61, 0x69, 0x6C, 0x65, 0x64, 0x2E, 0x20, 0x53, 0x65, 0x65, 0x20, 0x63, 0x75, 0x70, 0x6C, 0x54, 0x61, 0x67, 0x20, 0x64, 0x6F, 0x63, 0x73, 0x2E, 0xFE}

Hard-coded NDEF message containing one text record "Config check failed. See cuplTag docs."

const char **ndefmsg_badtrns**[] = {0x03, 0x27, 0xD1, 0x01, 0x23, 0x54, 0x02, 0x65, 0x6E, 0x45, 0x72, 0x72, 0x6F, 0x72, 0x3A, 0x20, 0x49, 0x6E, 0x76, 0x61, 0x6C, 0x69, 0x64, 0x20, 0x73, 0x74, 0x61, 0x74, 0x65, 0x20, 0x74, 0x72, 0x61, 0x6E, 0x73, 0x69, 0x74, 0x69, 0x6F, 0x6E, 0x2E, 0xFE}

Hard-coded NDEF message containing one text record "Error: Invalid state transition."

tretcode (***state_fcns**[])(*tevent*) = {*init_state*, *init_reqmemon*, *init_waitmemon*, *init_ntag*, *init_progmode*, *init_configcheck*, *init_errorcheck*, *init_wakeupcheck*, *init_batvwait*, *init_rtc_slow*, *init_rtc_1min*, *smpl_checkcounter*, *smpl_hdcreq*, *smpl_hdcwait*, *smpl_hdcread*, *smpl_wait*, *err_msg*, *end_state*}

struct *transition* **state_transitions**[]

The state transition table.

struct **transition**

Public Members

tstate **`src_state`**

Source state.

tretcode **`ret_code`**

Code returned after executing a state function.

tstate **`dst_state`**

Destination state.

2.2 HDC2021

A driver for the Texas Instruments HDC2021 temperature and humidity sensor.

Only the basic functionality is needed - to start the sensor in one-shot mode and collect a temperature and humidity measurement.

It is compatible with the HDC2022 and HDC2080 parts, which are electrically identical.

Defines

`TEMPL_REGADDR`

Temperature Low register address.

`TEMPH_REGADDR`

Temperature High register address.

`HUML_REGADDR`

Humidity Low register address.

`HUMH_REGADDR`

Humidity High register address.

`STAT_REGADDR`

Status register address.

`INTEN_REGADDR`

Interrupt enable register address.

`TEMPOFFSETADJ_REGADDR`

Temperature offset adjustment register address.

`HUMOFFSETADJ_REGADDR`

Humidity offset adjustment register address.

`DEVCONF_REGADDR`

Device configuration (soft-reset and interrupt reporting) register address.

`MEASCONF_REGADDR`

Measurement configuration register address.

`MANFIDL_REGADDR`

Manufacturer ID low-byte address.

`MANFIDH_REGADDR`

Manufacturer ID high-byte address.

`DEVIDL_REGADDR`

Device ID low-byte address.

`DEVIDH_REGADDR`

Device ID high-byte address.

`MEASCONF_MEAS_TRIG_BIT`

Trigger a measurement when applied to *MEASCONF_REGADDR*

`STAT_DRDY_STATUS_BIT`

Select the DRDY_STATUS bit from *STAT_REGADDR*

`INTEN_DRDYEN_BIT`

Enable the DataReady Interrupt when applied to *INTEN_REGADDR*

`DEVCONF_SOFT_RES_BIT`

Trigger a soft reset when applied to *DEVCONF_REGADDR*

`DEVCONF_DRDY_INTEN_BIT`

Enable the DRDY/INTEN pin when applied to *DEVCONF_REGADDR*

Functions

int **`hdc2021_startconv()`**

Trigger a one-shot conversion.

The HDC2021 is in one-shot mode. This function sends an I2C command to trigger the temperature and humidity measurement.

It also configures the interrupt pin on HDC2021 to make a high-to-low transition when the conversion data are ready. This takes ~500us, so the MSP430 can sleep in this time to save power.

int **`hdc2021_init()`**

Initialise the HDC2021 in one-shot mode.

uint32_t **`hdc2021_read_temprh`**(int *temp12b, int *rh12b)

Read temperature and humidity from the HDC2021.

Parameters

- **`temp12b`** – **[out]** Pointer to a variable for storing the raw 12-bit temperature.
- **`rh12b`** – **[out]** Pointer to a variable for storing the raw 12-bit relative humidity.

int **`hdc2021_read_whoami`**()

Read the Device ID registers.

Returns

A 16-bit Device ID. The expected value is 0x07D0.

2.3 NT3H

A driver for the NXP NT3H2111 NFC EEPROM.

Reads and writes memory on the NT3H2111 EEPROM using I2C. The memory is organised into 16-byte blocks from the I2C perspective.

BLOCK0 contains configuration data such as the Capability Container.

Defines

`ADDR_7b_MIN`

Minimum value of a 7-bit device address.

`ADDR_7b_MAX`

Maximum value of a 7-bit device address.

`WRONG_DEVADDR`

A wrong NFC EEPROM device address.

`BLOCK0`

Address of the first memory block.

`BLOCK_SESSION`

Address of the memory block used for session registers.

`DEVADDR_OFFSET`

Byte-offset of the Device Address within the first memory block.

`CC_OFFSET`

Byte-offset of the Capability Container within the first memory block.

`NSREG_OFFSET`

Byte-offset of the NS_REG session register.

`NSREG_EEPROM_WR_BUSY`

Select the EEPROM Write Busy bit from NS_REG. Set to 1'b1 when a write is in progress.

`BLOCKSIZE`

Block size in bytes.

`CC0_MAGIC`

Byte 0 of the Capability Container. Magic number.

`CC1_VER`

Byte 1 of the Capability Container. Version.

`CC2_NBYTESBY8`

Byte 2 of the Capability Container. The number of bytes in memory divided by 8.

Functions

void **`nt3h_init_wrongaddress`**(void)

Deliberately assign the wrong device address to the NFC EEPROM.

This can be called to re-create a problem where the NFC EEPROM ends up with the wrong device address.

Unfortunately, the device address is mutable and located in byte 0 of memory.

When power is running low, an unwanted transaction can occur (a series of zeroes) that clears the device address.

int **`nt3h_check_address`**(void)

Ensure that the NFC EEPROM is at the expected device address.

Check that the NFC EEPROM responds. If it does not, find it on the bus by scanning all available addresses. Once found, correct the address so there is no need to scan next time.

Check that the capability container is correct. If an external device has written to the tag, it may have been altered. It is not safe to write the capability container just-in-case. This will wear out the EEPROM block.

Returns

0 if OK or 1 if the Capability Container is not correct.

void **`nt3h_update_cc`**(void)

Write the Capability Container.

Sets the CC to the values required for long NDEF messages.

The CC is 0 from the factory. It can also be altered by a phone that writes to the tag.

This is a blocking operation. It will not return until the write is done.

int **`nt3h_writetag`**(int eepromBlock, char *blkdata)

Write one block to the NFC EEPROM.

Parameters

- **`eepromBlock`** – **[in]** index of the block to write
- **`blkdata`** – **[in]** 16-byte array containing data to write

Returns
A negative value if the write has failed.

int `nt3h_readtag`(int eepromBlock, char *blkdata)

Read one 16-byte block from the NFC EEPROM.

Parameters

- `eepromBlock` – **[in]** index of the block to read.
- `blkdata` – **[out]** 16-byte array into which the EEPROM block contents will be read.

void `nt3h_clearlock`(void)

Intended to clear the I2C_LOCKED bit in the NS_REG session register.

BUG. Does not affect NS_REG. Instead writes 0x06 to NC_REG, 0x40 to LAST_NDEF_BLOCK and 0x00 to SRAM_MIRROR_BLOCK.

Does not do any harm, because the device is powered down immediately afterwards anyway. Remove in future.

int `nt3h_eepromwritedone`(void)

Check if an EEPROM write is in-progress.

Returns
2 when a write is in progress. Otherwise 0 when EEPROM access is possible.

Variables

unsigned char `rxData`[BLOCKSIZE] = {0}

Holds the content of one block.

int `nsreg2`

int `nsreg3`

2.4 I2C

Defines

`HDC_DEVADDR`

HDC2021 I2C bus device address.

`NT3H_DEVADDR`

NFC EEPROM I2C bus device address.

Communicates with devices on an I2C bus.

Configures the EUSCI peripheral as an I2C master. Up to 16 bytes can be written to or read from a memory address on the I2C slave.

Author
Malcolm Mackay

Some devices embed up to 16 registers within each memory address. There is a function for reading one register only.

Defines

`EUSCI_BASE`

Base address of the EUSCI peripheral.

Functions

void **`i2c_init`**()

Initialise the EUSCI peripheral and I/O pins for I2C.

Weak pull-up resistors must be fitted to the I/O pins.

void **`i2c_off`**()

Put the EUSCI module into reset. Enable pull-downs on the I/O pins.

Floating pins waste power.

uint8_t **`i2c_write8`**(uint8_t sa, uint8_t mema, uint8_t txbyte)

Write one byte to the I2C device.

Blocks until the write has completed.

Parameters

- **`sa`** – **[in]** slave address
- **`mema`** – **[in]** memory address
- **`txbyte`** – **[in]** byte to write

int **`i2c_readreg`**(uint8_t sa, uint8_t mema, uint8_t rega)

Read one register on an I2C device.

Parameters

- **`sa`** – **[in]** slave address
- **`mema`** – **[in]** memory address
- **`rega`** – **[in]** register address

Returns

one byte of register data.

int **`i2c_write_block`**(uint8_t sa, uint8_t mema, uint8_t nbytes, uint8_t *txdata)

Write N bytes to the I2C device.

When NBYTES == 0:

| START | WRITE SA | MEMA | STOP |

When NBYTES >= 1:

| START | WRITE SA | MEMA | TXDATA[0] | TXDATA[…] | TXDATA[NBYTES-1] | STOP |

Parameters

- **`sa`** – **[in]** slave address.
- **`mema`** – **[in]** memory address.

- `nbytes` – **[in]** the number of bytes to write.
- `txdata` – **[in]** a pointer to the array of bytes to write.

int **`i2c_read_block`**(uint8_t sa, uint8_t mema, uint8_t nbytes, uint8_t *rxdata, uint8_t rega)

Read N bytes from the I2C device.

When a register address is specified:

| START | WRITE SA | MEMA | REGA | STOP | START | READ SA | BYTE0 | BYTE … | BYTE n-1 | STOP |

When no register address is specified:

| START | WRITE SA | MEMA | STOP | START | READ SA | BYTE0 | BYTE … | BYTE n-1 | STOP |

Parameters

- `sa` – **[in]** slave address.
- `mema` – **[in]** memory address.
- `nbytes` – **[in]** the number of bytes to read.
- `rxdata` – **[out]** a pointer to an array used to store read data. Must be at least nbytes long.
- `rega` – **[in]** register address. Set to 0xFF when a register read is not required.

Returns

-1 when the slave fails to respond, otherwise zero.

uint8_t **`i2c_read8`**(uint8_t sa, uint8_t mema)

Read one byte from memory on the I2C slave.

START | WRITE SA | MEMA | STOP | START | READ SA | BYTE0 | STOP.

Parameters

- `sa` – **[in]** slave address.
- `mema` – **[in]** memory address.

Returns

one byte read from the I2C slave.

uint16_t **`i2c_read16`**(uint8_t sa, uint8_t mema)

Read two bytes from memory on the I2C slave.

START | WRITE SA | MEMA | STOP | START | READ SA | BYTE0 | BYTE1 | STOP.

Parameters

- `sa` – **[in]** slave address.
- `mema` – **[in]** memory address.

Returns

one little-endian 16-bit integer read from the I2C slave.

uint16_t **`i2c_read16x2`**(uint8_t sa, uint8_t mema, uint16_t *uint0, uint16_t *uint1)

Read two consecutive unsigned integers from the I2C slave.

START | WRITE SA | MEMA | STOP | START | READ SA | BYTE0 | BYTE1 | BYTE2 | BYTE3 | STOP.

Parameters

- `sa` – **[in]** slave address.
- `mema` – **[in]** memory address.

- **uint0** – **[out]** pointer to an address for storing the first little endian unsigned integer.
- **uint1** – **[out]** pointer to an address for strong the second little endian unsigned integer.

void **USCIB0_ISR**(void)

Variables

uint8_t **buffer**[16] = {0}

Read or write buffer. This is declared volatile because it is accessed from an ISR.

uint8_t **bytesLength** = 0

Transaction length. This is declared volatile because it is read from an ISR.

uint8_t **gbl_regOffset** = 0

Memory address. Volatile because it is read from an ISR.

bool **restartTx** = false

bool **nackFlag** = false

bool **stopFlag** = false

bool **restartRx** = false

2.5 Stat

Functions

void **stat_rdrstcause**()

Find what has caused the latest reset by reading the System Reset Interrupt Vector register.

This function sets bits in the global variable rstcause, which is initialised to zero. Only the highest priority interrupt is read.

int **stat_rstcause_is_lpm5wu**()

Check if the reset was caused by a routine wakeup from LPMx.5.

LPMx.5 is entered each time the state machine runs during normal operation. This is to minimise power consumption. The Program Counter resets to zero and RAM is powered down. The Real Time Clock (RTC) peripheral triggers a reset (and an exit from LPMx.5) after one minute has elapsed.

The *stat_rdrstcause()* function must be called first.

Returns

Non-zero when the reset has been caused by a wake-up from LPMx.5

void **stat_setclockfailure**()

Set the clock failure bit in the rstcause global variable.

unsigned int `stat_get`(bool *err, bool *borsvs, int resetsalltime)

Get status information from the rstcause global variable.

Parameters

- `err` – **[out]** Pointer to an error flag. The flag is set if the latest reset has been caused by an error.
- `borsvs` – **[out]** Pointer to the Brownout or SVS reset flag. The flag is set if the latest reset has been caused by a voltage drop to the SVSH or BOR levels.
- `resetsalltime` – **[in]** Number of resets that have occurred from the factory.

Returns

A 16-bit status word for inclusion in the URL by cuplCodec. The upper byte is resetsalltime/16. The lower byte is a copy of the rstcause variable.

Variables

unsigned int `rstcause` = 0

Reset cause global variable.

2.6 Config NFC

Defines

`EEPROM_USERMEM_FIRST_BLOCK`

Index of the first 16-byte block of unprotected user memory.

`NDEF_RECORDTYPE_TEXT`

NDEF text record type.

`PAYLOADSTART_SHORTREC_INDEX`

NDEF record payload starts at this byte within EEPROM block 1.

`RECORDTYPE_SHORTREC_INDEX`

Index corresponding to NDEF record type. Only correct for short NDEF records.

`PAYLOADLEN_SHORTREC_INDEX`

Index corresponding to NDEF record length. Only correct for short NDEF records.

`CONFIGSTR_STARTCHAR`

Marks the start of a configuration string.

`CONFIGSTR_DELIMCHAR`

Separates the ID from a value in a configuration string.

`CONFIGSTR_ENDCHAR`

Marks the end of a configuration string.

Enums

enum **`parserstate_t`**

Values:

enumerator **`findstartchar`**

Search for the config string start character.

enumerator **`storeid`**

Read the ID byte.

enumerator **`checkdelimiter`**

Check for the delimiter byte.

enumerator **`storevalue`**

Copy the configuration string value into msgblock.

Functions

int **`confignfc_check`**`()`

Read the first block of NFC EEPROM user memory. Check if it contains an NDEF text record.

Configuration data are written as strings in the text record.

Returns
1 if a short text record is present, otherwise 0.

int **`confignfc_parse`**`()`

Parse the NDEF text record into configuration strings. Writing configuration to NVM.

Configuration strings are formatted as:

<c:xyz>

Where:

‘<’ is the start character.

‘c’ is the ID.

‘:’ is the delimiter.

‘xyz’ is the value.

‘>’ is the end character.

The text record can contain one or more strings. There is no separator character between them.

Configuration strings are written to non-volatile memory with *nvparams_write()*.

Variables

static char **readbuffer**[BLKSIZE]

Holds one 16-byte EEPROM block.

unsigned char **msgblock**[64]

Re-use an array declared as part of cuplcodec for calculating the MD5 checksum.

2.7 Comms UART

Defines

UART_BAUDRATE

Typedefs

typedef enum *ustat* **t_ustat**

Enums

enum **ustat**

Values:

enumerator **ustat_running**

enumerator **ustat_waiting**

enumerator **ustat_finished**

Functions

t_ustat **uart_run**()

Defines

HW_VERSION

FW_VERSION

XSTR(V)

STR(V)

VERSION

EXIT_STATE

ENTRY_STATE

INDEX_ID

INDEX_VAL

Typedefs

typedef enum *uart_ret_codes* **t_uretcode**

typedef enum *uart_event_codes* **t_uevent**

typedef enum *uart_state_codes* **t_ustate**

Enums

enum **uart_ret_codes**

Values:

enumerator **rc_ok**

enumerator **rc_fail**

enumerator **rc_wait**

enum **uart_event_codes**

Values:

enumerator **evt_none**

enumerator **evt_rxdone**

enumerator **evt_txdone**

enum **uart_state_codes**

Values:

enumerator **uartsc_init**

enumerator **uartsc_txboot**

enumerator **uartsc_prepRx**

enumerator **uartsc_waitforRx**

enumerator **uartsc_pcktrxed**

enumerator **uartsc_prepTx**

enumerator **uartsc_waitforTx**

enumerator **uartsc_error**

Functions

t_uretcode **uart_init**(*t_uevent* evt)

t_uretcode **uart_txboot**(*t_uevent* evt)

t_uretcode **uart_prepRx**(*t_uevent* evt)

t_uretcode **uart_waitforRx**(*t_uevent* evt)

t_uretcode **uart_pcktrxed**(*t_uevent* evt)

t_uretcode **uart_prepTx**(*t_uevent* evt)

t_uretcode **uart_waitforTx**(*t_uevent* evt)

t_uretcode **uart_error**(*t_uevent* evt)

static *t_ustate* **lookup_transitions**(*t_ustate* curstate, *t_uretcode* rc)

t_ustat **uart_run**()

__interrupt void USCI_A0_ISR (void)

Variables

static char __version__ [] = "<x:" VERSION ">"

static *t_uretcode* (***ustate_fcns**[])(*t_uevent*) = {*uart_init*, *uart_txboot*, *uart_prepRx*, *uart_waitforRx*, *uart_pcktrxed*, *uart_prepTx*, *uart_waitforTx*, *uart_error*} struct *utransition* **ustate_transitions**[] = {{uartsc_init, rc_ok, uartsc_txboot}, {uartsc_txboot, rc_ok, uartsc_prepTx}, {uartsc_prepRx, rc_ok, uartsc_waitforRx}, {uartsc_waitforRx, rc_ok, uartsc_pcktrxed}, {uartsc_waitforRx, rc_wait, uartsc_waitforRx}, {uartsc_pcktrxed, rc_ok, uartsc_prepTx}, {uartsc_pcktrxed, rc_wait, uartsc_pcktrxed}, {uartsc_prepTx, rc_ok, uartsc_waitforTx}, {uartsc_waitforTx, rc_ok, uartsc_prepRx}, {uartsc_waitforTx, rc_wait, uartsc_waitforTx}}

uint8_t **uartBuffer**[72]

unsigned int **bufIndex** = 0

int **drdyFlag** = 0

int **txDoneFlag** = 0

static *t_ustate* **cur_state** = ENTRY_STATE

struct **utransition**

Public Members

t_ustate **src_state**

t_uretcode **ret_code**

t_ustate **dst_state**

2.8 Non-Volatile Parameters

Reads and writes parameters in non-volatile memory (FRAM).

The cuplTag (and cuplcodec) is configured with a small set of parameters. These control the tag serial string, the sampling interval or the URL of the web application that decodes the tag contents.

A parameter has a single-byte identifier e.g. ‘w’ and a value. The length of the value field depends on the parameter. For example, the serial string consists of eight bytes (e.g. ‘AB43xkp4’).

A variable is used to monitor how many parameters have been written since the last power cycle. A full set of parameters is needed for the program to proceed.

Defines

`NVPARAM_SERIAL_ID`

Serial ID

`NVPARAM_SECKEY_ID`

Secret key ID

`NVPARAM_BASEURL_ID`

Base URL of the cupl web application ID

`NVPARAM_FMT_ID`

Sample format ID

`NVPARAM_SMPLINT_ID`

Sample interval ID

`NVPARAM_MINVOLT_ID`

Minimum operating voltage (in mV) ID

`NVPARAM_HTTPSDIS_ID`

Disable HTTPS ID

`NVPARAM_USEHMAC_ID`

Use HMAC ID

`SERIAL_PARAM_WRITTEN`

`SECKEY_PARAM_WRITTEN`

`FMT_PARAM_WRITTEN`

`SMPLINT_PARAM_WRITTEN`

`MINVOLT_PARAM_WRITTEN`

`BASEURL_PARAM_WRITTEN`

`HTTPSDIS_PARAM_WRITTEN`

`USEHMAC_PARAM_WRITTEN`

`ALL_PARAMS_WRITTEN`

0xFF in the ‘paramswritten’ RAM variable indicates that all parameters have been written.

`NVM_ALL_PARAMS_WRITTEN`

Zero in NVM indicates that all parameters have been written. Why zero? After programming, the initial value for this NVM section is 0xFF.

`MINUTES_PER_DAY`

`INTEGERFIELD_LENBYTES`

Value up to 65535 (5 ASCII digits)

`DISABLE_FRAM_DATA_WRITEPROTECT`

Clear the Data FRAM Write Protect bit.

`ENABLE_FRAM_DATA_WRITEPROTECT`

Set the Data FRAM Write Protect bit.

Functions

char ***nvparams_getserial**()

Get the 8-character alphanumeric serial string.

This is used to identify a cuplTag to the server. It is included in the URL.

Returns
A pointer to the serial string.

char ***nvparams_getsecretkey**()

Get the secret key used by cuplcodec to calculate an HMAC-MD5.

The secret key is unique per tag. It is known only to the web server and the tag. It is used to generate a Hash based Message Authenticity Code, which prevents an 'imposter tag' from writing sample data to the web server.

Returns
A pointer to the secret key, which is SECKEY_LENBYTES long.

unsigned int **nvparams_getminvoltagemv**(void)

Get the minimum operating voltage parameter.

If the battery voltage is allowed to drop to the brown-out voltage, then the NFC EEPROM will be left with stale data. There will be insufficient power to overwrite this with a 'low power' message, because the MSP430 will be stuck in a reset loop.

This parameter should be slightly higher than the brown-out voltage e.g. 2200mV. When it is reached, the sampling loop will stop, sample data removed and the user will be notified.

Returns
Minimum operating voltage in millivolts.

unsigned int **nvparams_getsmplintmins**()

Get the sample interval in minutes.

Temperature/humidity sensor samples are written to the circular buffer at this interval.

Returns
The sample interval in minutes as a 16-bit unsigned integer.

long **`nvparams_getsleepintmins`**`()`

Get the sleep interval in minutes (deprecated).

This NVM parameter is not used.

Returns

The tag sleep interval in minutes.

bool **`nvparams_allwritten`**`()`

Check that the cuplTag is fully configured.

Returns

'true' if all NVM parameters have been written.

int **`nvparams_getresetsperloop`**`()`

Get resets per loop.

Returns

The number of resets that have occurred during the present 'loop' of the circular buffer.

int **`nvparams_getresetsalltime`**`()`

Returns

The number of resets that have occurred from the factory.

void **`nvparams_cresetsperloop`**`()`

Clear resets per loop.

Clear the number of resets that has occurred during the present 'loop' of the circular buffer. This should be done each time data wraps from the end of the circular buffer back to the start.

void **`nvparams_incrcounters`**`()`

Increment both reset counters in NVM.

bool **`nvparams_write`**`(`char id, char *valptr, unsigned int vlen`)`

Write a parameter to non-volatile memory.

A parameter consists of an ID (one byte) and a value (one or more bytes).

Parameters

- **`id`** – **[in]** parameter ID.
- **`valptr`** – **[in]** pointer to an array that contains the parameter value.
- **`vlen`** – **[in]** length of the value in bytes.

Returns

true if the parameter ID is recognised and its value has the correct length in bytes. Otherwise false.

Variables

nv_t **`nv`**

unsigned int **`paramswritten`** = 0

Bits are set in this integer that correspond to parameters written since the last power cycle.

2.9 Battery Voltage

Functions

static void **`adc_enable`**()

static void **`adc_disable`**()

unsigned int **`batv_measure`**()

unsigned int **`batv_to_mv`**(unsigned int batv)

void **`ADC_ISR`**(void)

Variables

unsigned int **`adcvoltage`** = 0

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX