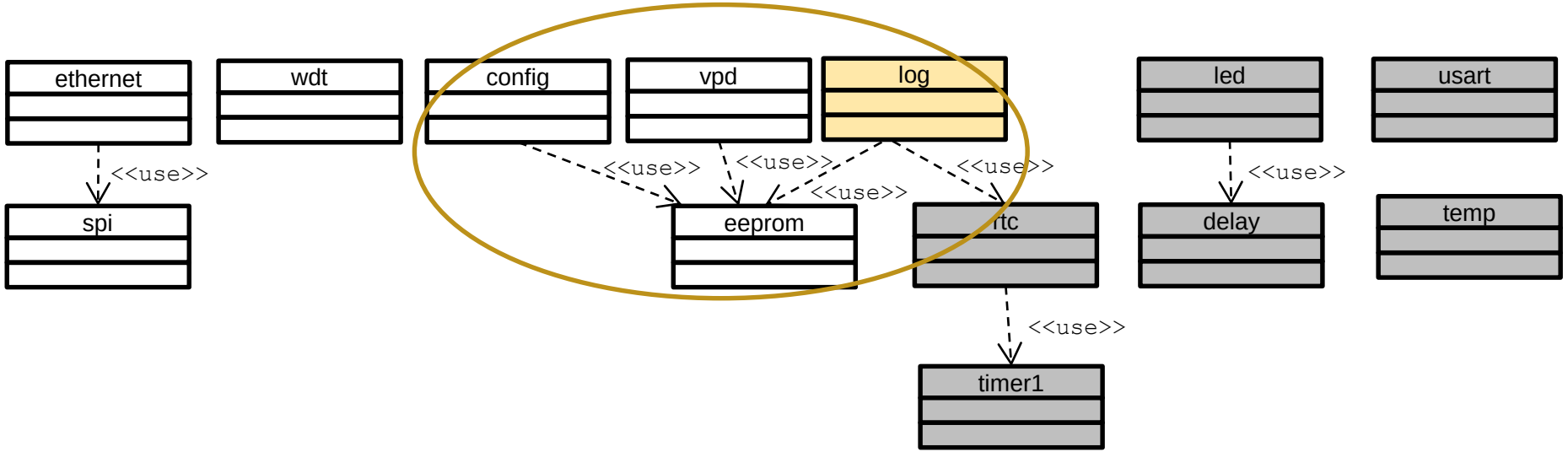
SER486: Embedded C Programming

Design Specification

Programming Project 2
EEPROM/Vital Product & Configuration Data/ System Event Log

Device Class Diagram



This project focuses on the circled classes.

Classes shown in grey have been completed already and are part of the project 2 library code

EEPROM class

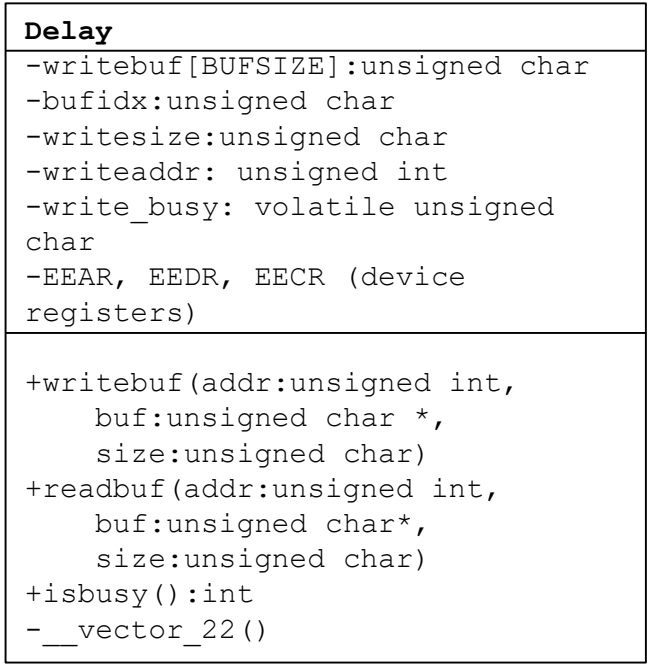
The eeprom class implements a hardware device design pattern to encapsulate reading/writing of the ATMEGA 328P EEPROM. Reads are read directly from the device. Writes are buffered and written using an interrupt service routine. More information on this device can be found in the ATMEGA 328P datasheet.

Internal State

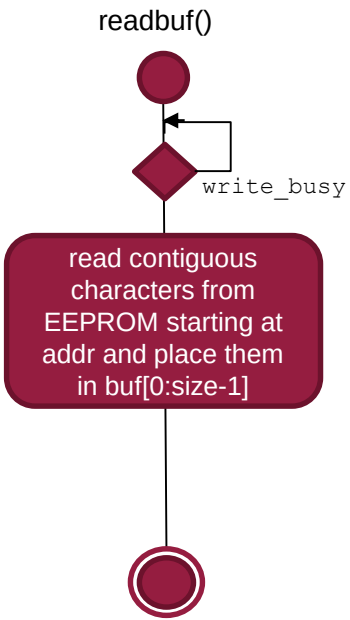
- **writebuf[]** – A 64-byte write buffer used by writebuf() and the interrupt service routine.
- **bufidx** – The index of the next writebuf character the interrupt service routine should write.
- **writesize** – The size of the data (in bytes) within the write buffer that must be written.
- **writeaddr** – Used by the interrupt service routine, specifies the next EEPROM address with which to write the data to.
- **write_busy** – Set to 1 if there is data in the write buffer that needs to be written. Cleared to 0 by the interrupt service routine when the last byte of the data has been written.

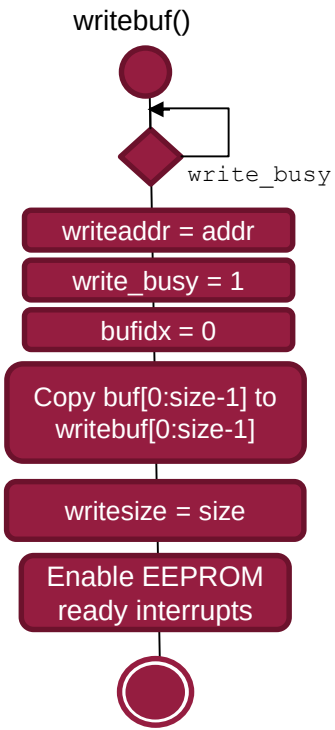
Member Functions

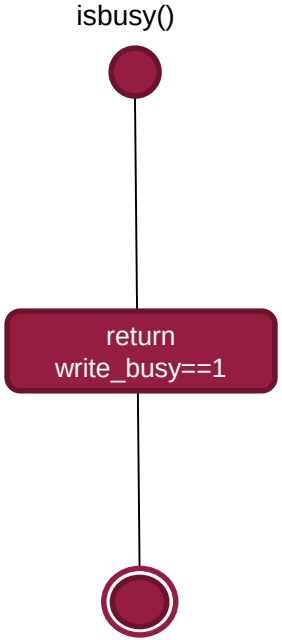
- **writebuf()** – places the data (specified by buf and size) into the write buffer for later writing to the EEPROM. The addr parameter specifies the location to write the data to. This function should not be called when another write is in progress.
- **readbuf()** – reads a specified amount of data (size) from the EEPROM starting at a specified address (addr) and places it in the specified buffer (buf).
- **isbusy()**– returns 0 if write_busy is 0, otheriwise, returns 1.
- **void __vector_22()** – Enabled when writebuf() places new data in the write buffer, this ISR sends one byte at a time to the EEPROM. When the last byte is sent, it disables further EEPROM interrupts.

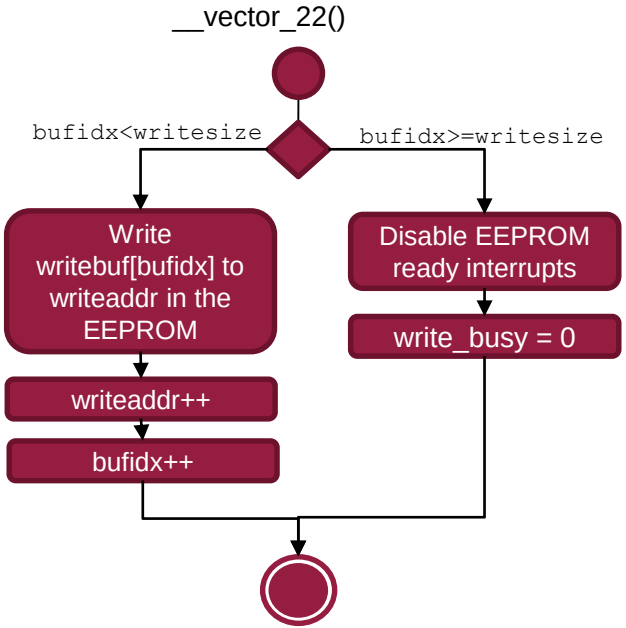


EEPROM methods



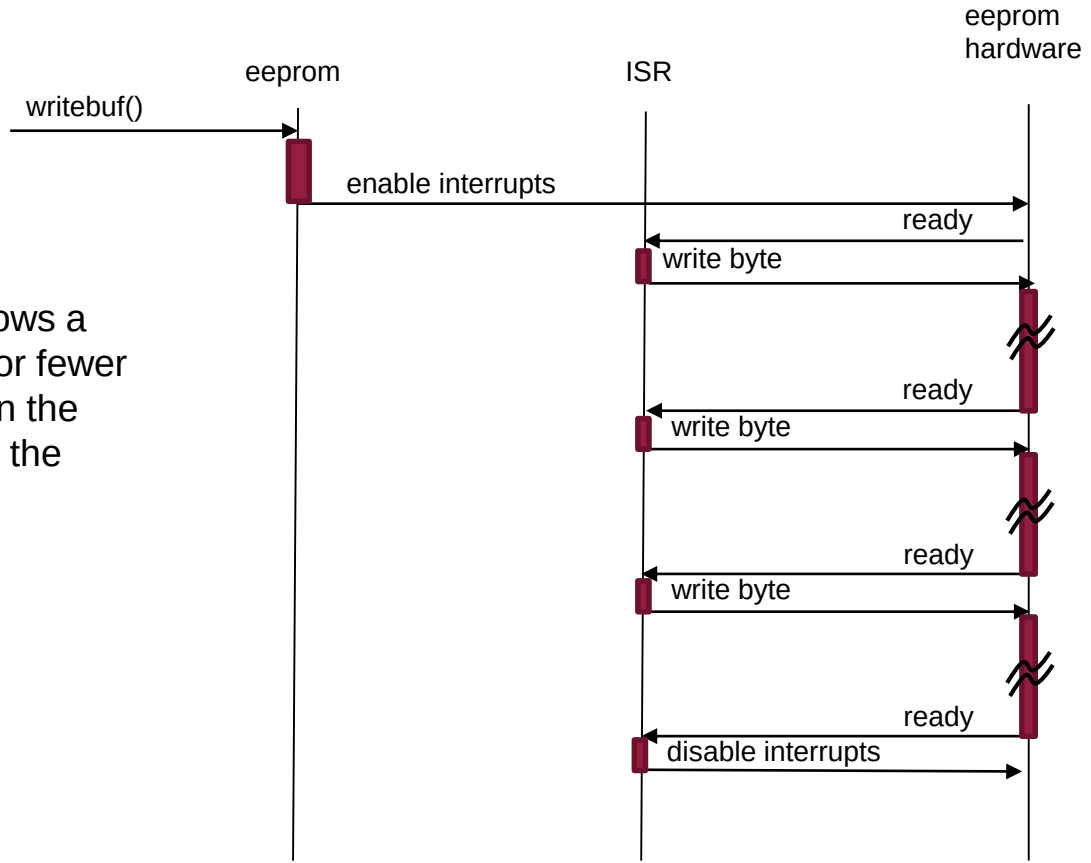






Interrupt Handler for EEPROM

This sequence diagram shows a write of three bytes. More or fewer bytes are possible based on the size of the data provided to the writebuf() function.



VPD class

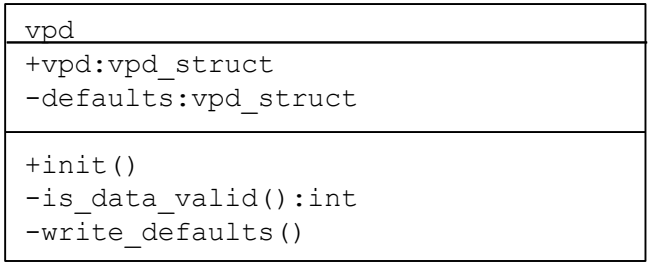
The vpd class implements non-volatile vital product data for the IIoT device. Data is validated through use of a string token identifier and data checksum. Upon data corruption (or first time use), data will be initialized to "Factory Defaults". The utility functions update_checksum() and is_checksum_valid() are used to write and verify the vpd checksum.

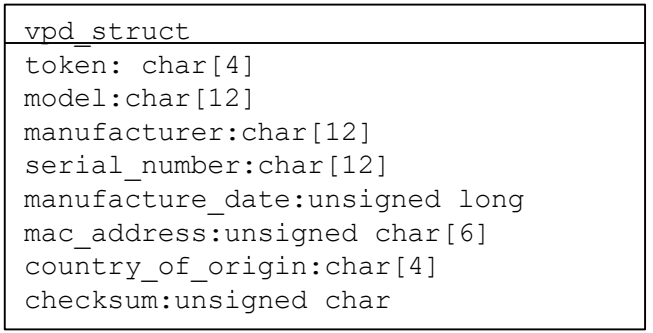
Internal State

- **vpd –** the vital product data read from the Eeprom at address 0x000. This data is represented as a structure and individual members can be accessed using dot notation (e.g. vpd.serial_number).
- **defaults** – the factory defaults for the vpd. (see next page)

Member Functions

- **init()** – initializes vpd member data from the EEPROM. If vpd data is invalid after initialization, the EEPROM is written to "factory defaults", and the vpd data is reinitialized from the new EEPROM values.
- **is_data_valid()** – returns 1 (true) if the vpd token is "SER" and the checksum is valid. Otherwise, returns 0.
- **write_defaults()**– called by init() to write the "factory defaults" to the EEPROM.





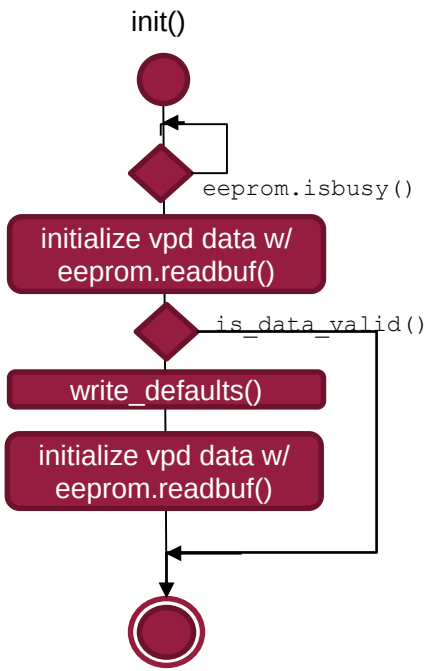
Factory Default Values for VPD

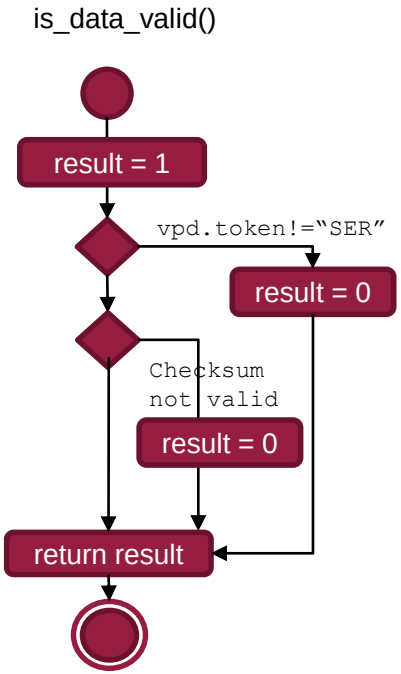
Use structure initialization syntax to set the following values for defaults

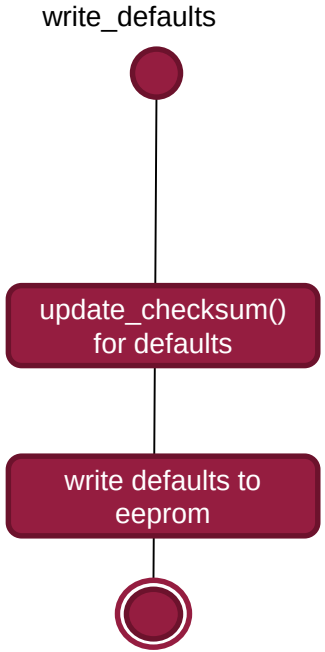
The VPD structure within the EEPROM should begin at address 0x0000

Field	Value	
.token	“SER”	
.model	Your first name	No more than 11 characters
.manufacturer	Your last name	No more than 11 characters
.serial_number	Any string	No more than 11 characters
.manufacture_date	0	
.mac_address	{‘x1’,’x2’,’x3’,’y1’,’y2’,’y3}	x1 = ascii code for the first letter in your name anded with 0xFE x2-x3 = asci codes for the second and third letters of your first name y1-y3 = asci codes for the first three letters of your last name
.country_of_origin	“USA”	
.checksum	0	

VPD methods


init()
eeprom.isbusy()
initialize vpd data w/ eeprom.readbuf()
is_data_valid()
write_defaults()
initialize vpd data w/ eeprom.readbuf()


is_data_valid()
result = 1
vpd.token!="SER"
result = 0
Checksum not valid
result = 0
return result


write_defaults
update_checksum() for defaults
write defaults to eeprom

CONFIG class

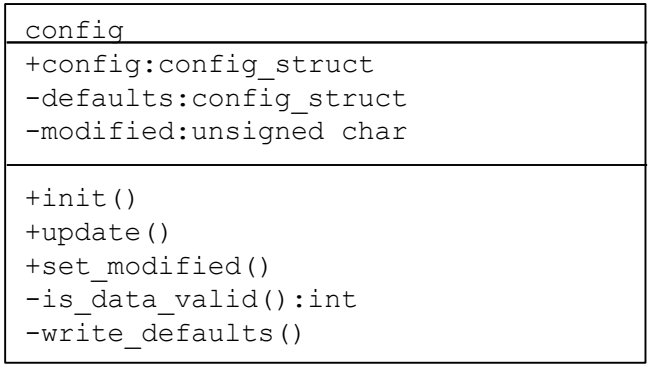
The config class implements non-volatile configuration data for the IIoT device. Data is validated through use of a string token identifier and data checksum. Upon data corruption (or first time use), data will be initialized to "Factory Defaults". The utility functions update_checksum() and is_checksum_valid() are used to write and verify the configuration checksum.

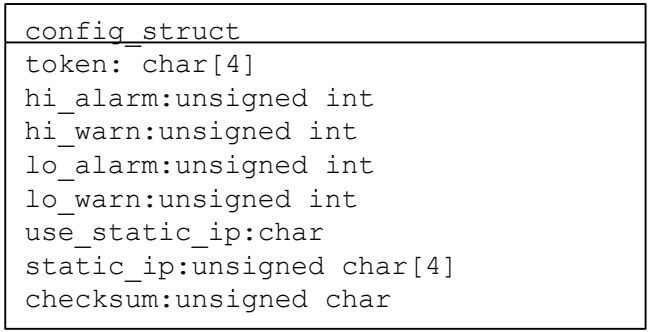
Internal State

- **config –** the configuration data read from the Eeprom at address 0x040. This data is represented as a structure and individual members can be accessed using dot notation (e.g. config.static_ip). The data also acts as a write cache. Data may be read/written while EEPROM updates are in progress.
- **defaults** – the factory defaults for config. (see next page)
- **modified** – set by the set_modified() function, this variable signals that the data has been modified and should be written to eeprom during the next update cycle.

Member Functions

- **init()** – initializes config member data from the EEPROM. If config data is invalid after initialization, the EEPROM is written to "factory defaults", and the config data is reinitialized from the new EEPROM values.
- **update()** – if config has been modified and eeprom is not busy, write all of the configuration data to the eeprom write buffer.
- **set_modified()** – set the modified flag. This function should be called any time the config data is modified.
- **is_data_valid()** – returns 1 (true) if the token is "ASU" and the checksum is valid. Otherwise, returns 0.
- **write_defaults()**– called by init() to write the config "factory defaults" to eeprom.





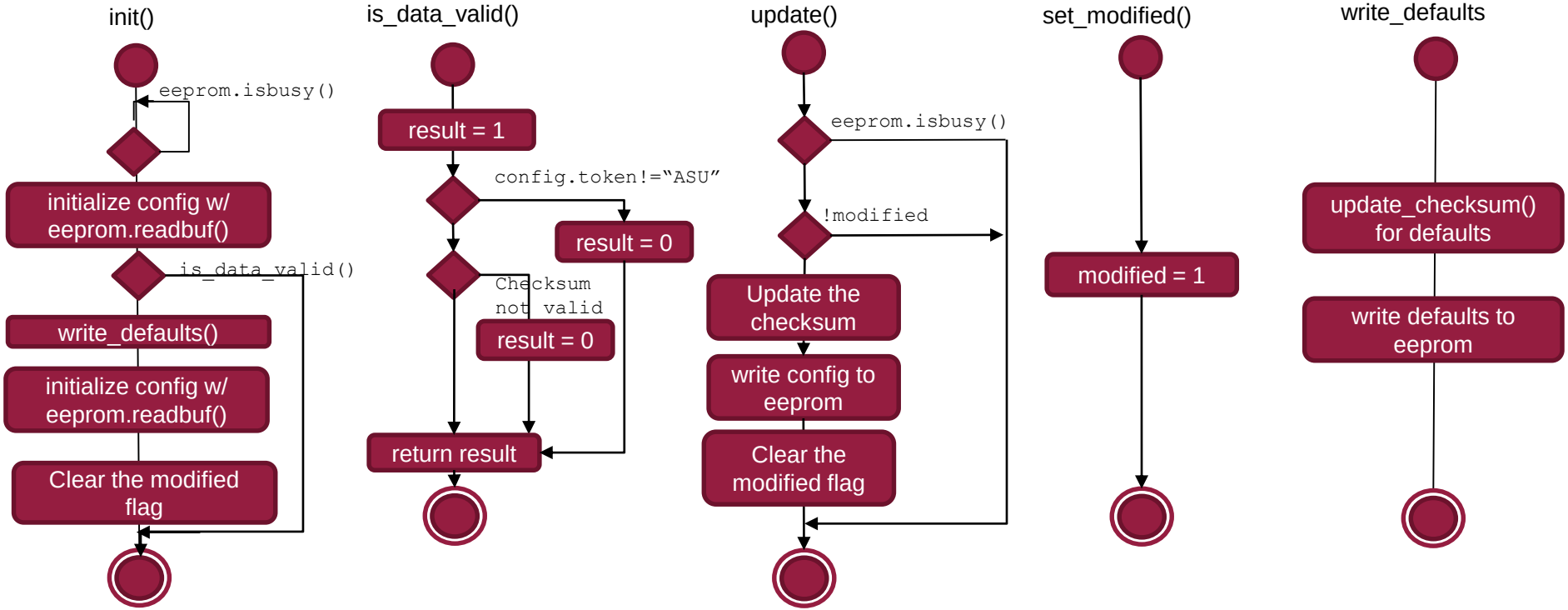
Factory Default Values for Config

Use structure initialization syntax to set the following values for defaults

The Config structure within the EEPROM should begin at address 0x0040

Field	Value
.token	“ASU”
.hi_alarm	0x3FF
.hi_warn	0x3FE
.lo_alarm	0x0000
.lo_warn	0x0001
.use_static_ip	0
.static_ip	{192,168,1,100}
.checksum	0

Config methods

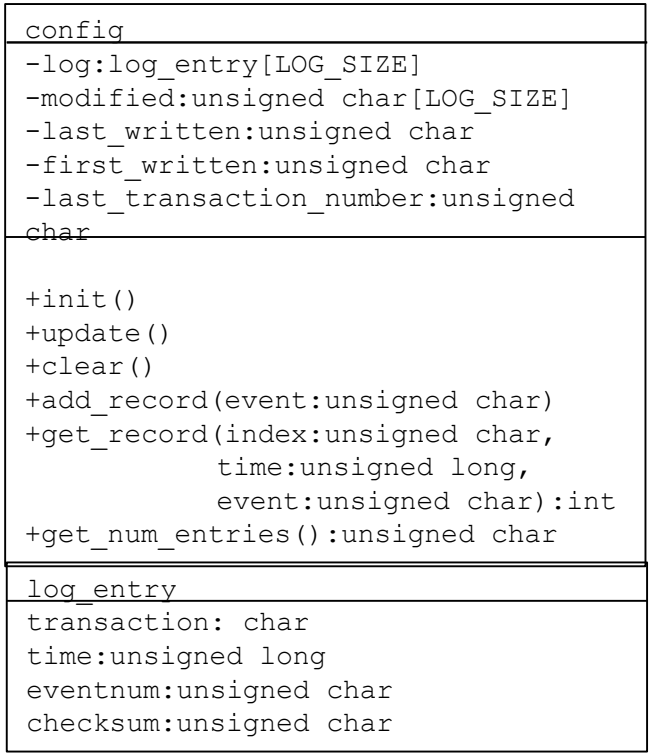


Log class (instructor Provided)

The log class implements non-volatile system event log for the IIoT device. Data is validated through use of a data checksum. The log holds 16 entries and is implemented as a circular queue with overwrite.

Public Member Functions

- **init()** – initializes the log_entries by reading eeprom. Based on transaction numbers, determines the most and least recent log entries (identifying the front and rear of the queue).
- **update()** – If eeprom is not busy, this function searches through the log entries and writes back the first modified entry that it finds (setting the modified flag to 0 in the process)
- **clear()** – clears the log so that there are no entries. Sets the modified flag on all entries to force a writeback to eeprom on next update.
- **add_record()** – add a new record to the log (replacing oldest record if necessary). The type of event is specified by the event parameter. The modified bit is set for the log entry so that it will be written back when an update occurs.
- **get_record()** – returns the log record specified by index (index 0 is the oldest log entry). The log event and time are returned in the time and event function parameters. If the log entry does not exist, the function returns 0. Otherwise, it returns 1.
- **get_num_entries()** – returns the number of valid entries in the log.



Utility functions (util.c / util.h)

update_checksum(char*data, unsigned int size)

- Parameters:
  - data – a pointer to the data structure to update the checksum
  - size – the size of the datastructure (use sizeof() to pass this parameter)
- Description
  - This function changes the last byte of the data structure so that the sum of all the bytes within the data structure will be zero.

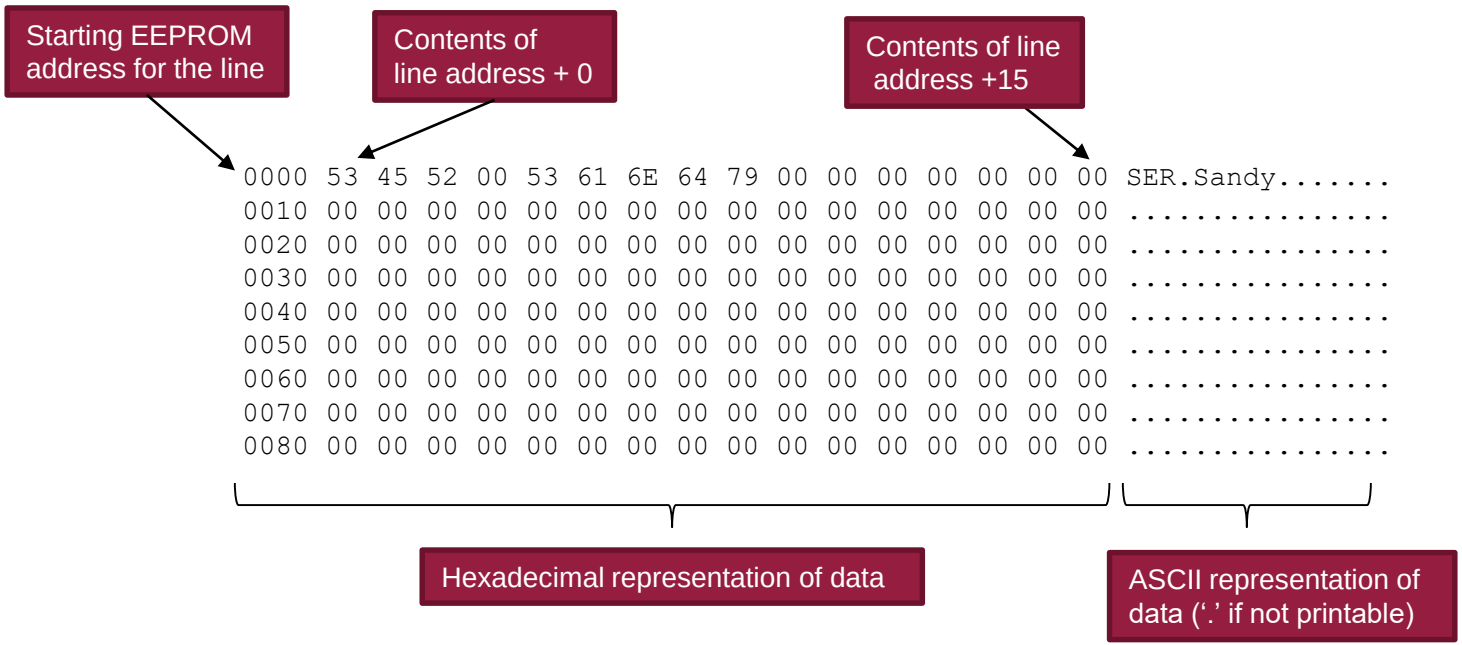
is_checksum_valid(char*data, unsigned int size):int

- Parameters:
  - data – a pointer to the data structure to validate
  - size – the size of the datastructure (use sizeof() to pass this parameter)
- Return:
  - 1 if the checksum is valid, 0 otherwise.
- Description
  - This function sums all the byte values within the structure and returns 1 if the sum is zero.

dump_eeprom(unsigned int addr, unsigned int numbytes) (INSTRUCTOR PROVIDED)

- Parameters:
  - address – the address of the first byte to display to the console.
  - numbytes – the number of bytes to display to the console.
- Description:
  - Dumps the contents of the eeprom from address to address+numbytes to the console. Both hexadecimal and asci representations are shown. For format, see next slide.

Dump_EEPROM Format




/* BrainFuck.c
* BrainFuck Interpreter */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int main(int argc, char **argv)
{
int memory[32768], space[32768];
int fd, len, buf, spointer, mpointer;
int kl = 0;
if (argc <= 1)
{
printf("Usage: %s <brainfuck file>\n", argv[0]);
exit(1);
}
/* Open brainfuck file */
if ((fd = open(argv[1], O_RDONLY)) < 0) exit(1);
/* Read the file byte by byte */
len = spointer = 1;
while (len > 0)
{
len = read(fd, &buf, 1);
space[spointer] = buf;
spointer++;
}
close(fd);
for (mpointer = 0; mpointer < 32768; mpointer++) memory[mpointer] = 0;
len = spointer;
spointer = mpointer = 0;
for (spointer = 0; spointer < len; spointer++)
{
switch(space[spointer])
{
// Increment pointer value
case '+':
memory[mpointer]++;
break;
// Decrement pointer value
case '-':
memory[mpointer]--;
break;
// Increment pointer
case '>':
mpointer++;
break;
// Decrement pointer
case '<':
mpointer--;
break;
// Print current pointer value
case '.':
putchar(memory[mpointer]);
break;
// Read value and store in current pointer
case ',':
memory[mpointer] = getchar();
break;
// Start loop
case '[':
if (memory[mpointer] == 0)
{
/* Find matching] */
spointer++;
while (kl > 0 || space[spointer] != ']')
{
if (space[spointer] == '[') kl++;
if (space[spointer] == ']') kl--;
// Go in right direction
spointer++;
}
}
break;
// End Loop
case ']':
if (memory[mpointer] != 0)
{
/* Find matching [*/
spointer--;
while (kl > 0 || space[spointer] != '[')
{
if (space[spointer] == ']') kl++;
if (space[spointer] == '[') kl--;
// Go left
spointer--;
}
spointer--;
}
break;
}
}
putchar('\n');
return (0);
}

[END]