RecordBOX: A Framework for a Lightweight Record Oriented Storage System For IoT Sensor Nodes

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

01-03-2021 / 03-03-2021

CITATION

Bhavsar, Yagnik; Patel, Chintan (2021): RecordBOX: A Framework for a Lightweight Record Oriented Storage System For IoT Sensor Nodes. TechRxiv. Preprint.
https://www.techrxiv.org/articles/preprint/RecordBOX_A_Framework_for_a_Lightweight_Record_Oriented_Stora

RecordBOX: A Framework for a Lightweight Record Oriented Storage System For IoT Sensor Nodes

Yagnik Bhavsar *Member, IEEE*, Chintan Patel, *Member, IEEE*,

Abstract— Sensor nodes are widely used in smart agriculture, smart home, industrial automation, healthcare monitoring, and many more. Each sensor node has sensors connected to it, and this sensor node acquires environment data using these sensors and sends it to the base station or gateway. The backbone of such a system is connectivity, where each sensor node must be connected with the base station via wireless technologies such as Bluetooth, ZigBee, or WiFi. These sensor nodes have small storage because they are built to send data continuously rather than storing it. In such a scenario where connectivity fails, the sensor node requires to store these critical data. For that matter, the storage system must be lightweight because sensor nodes are constrained devices. In this paper, we propose a novel framework for record-oriented storage system for wireless sensor nodes. The proposed approach has a tiny memory footprint, and that is very useful for memory-constrained sensor nodes. It can be built for any byte-addressable memory. Though it is a record-oriented storage system, we store data in a predefined structure only. In this paper, we propose a unique framework of the storage system with a simple user interface. The proposed framework automatically generates a record and library based on the user's memory, size, type of the sensors, and language that is a lightweight and efficient for sensing nodes.

Index Terms—storage system, sensor node, sensor network, EEPROM memory.

I. INTRODUCTION

We can classify the sensor nodes into two categories as per their transmission rates of data. In the first category, the sensing node continuously transmits the data, and in the second one, it sends data periodically. Each sensor node reads data from the number of connected sensors, and transfer it to the base station or stores it for later transmission if connectivity fails. Also, while the receiver and transmitter are working on different data rates or the base station is busy, in such a situation, we need a buffer to store the data transitorily. In specific Wireless sensor network (WSN) application, all sensor nodes carry the same group of data, notwithstanding the technologies they are using for the data transfer or underlying controller. In a critical application such as industrial automation and the health-care, we need endless monitoring of the data. While in other applications such as smart agriculture, we

need periodic data collection. In both cases, connectivity loss may occur anytime during the data transfer, and it is highly undesirable to lose the data generated during the connectivity issue. Therefore, we need a storage buffer at sensor node to store the sensed data in such a scenario. It is unwise to deploy the complex and frugal file system on the sensor node for holding data momentarily. Hence, to solve this problem, we propose a framework that provides an efficient, fully optimized and computationally lightweight design of the storage system, that can be implemented on the sensing node.

Figure 1. Typical Wireless Sensor Network, [1]

Figure 1 highlights the basic structure of a typical WSN application. The sensors are layer-1 devices which are deployed on the field for data collection. These sensors transmit collected data to the base station through the gateway device. The "base station" is a resource capable device and can process the massive data speedily and steadily. In certain conditions (such as electricity fault or Denial of Service (DoS) attack), while the gateway or base station might not promptly respond to the sensor node, we need a buffer to store the sensed data.

Table I provides comparative analysis for various sensor nodes available in market. These sensor nodes are widely used and adopted by the WSN and Internet of Things (IoT) industries for data collection. Most of the sensor nodes having the support of certain operating systems (OS) [2] such as TinyOS [3], [4], [5], LiteOS [6], [7], and Contiki [8], [9]. The crossbow has developed IMote2 [10] that uses Intel PXA271 Xscale processor. The CDAC also developed Ubi-Mote1, and Ubi-

Yagnik is teaching associate with the Department of Information and communications technology, Pandit Deendayal Petroleum University, Gandhinagar, India.

E-mail: bhavsaryagnik003@gmail.com

Chintan Patel is corresponding author of the manuscript and doctorate student with the Department of Computer Science and Engineering, Pandit Deendayal Petroleum University, Gandhinagar, India.

E-mail: chintan.p592@gmail.com

Mote2 with TI CC2430 SOC and TI CC2520 micro controller respectively [11]. Table II put forward comparison between different serial memory systems developed by microchip. It is apparent that the fastest memory is SRAM, but it is a volatile memory. While FLASH and EEPROM have similar read/write cycle time but, standby current required by EEPROM is lesser compared to FLASH. Therefore, for power-constrained sensor nodes, EEPROM can be an ideal choice for the data storage purpose.

The EEPROM memory works in a byte-addressable mode means a single byte in EEPROM can read and write at a time. At the same time, FLASH performs the read-write operation in a block-wise manner. The "Endurance" of memory is the guaranteed number of times the given type of memory can be reprogrammed and still operate across the given conditions (temperature, voltage). The EEPROM has higher endurance than FLASH memory. Usually, the FLASH is used to store program code on the device, and EEPROM is used to store the data. It shows that EEPROM can also work as a tiny hard disk. Different types of SRAM memories such as NVSRAM and EERAM can be used for storage, but they are not available for microcontrollers. The proposed "RecordBOX" works for them as well.

Motivation: There are several file systems already developed for embedded platforms. All such systems use a file as an abstraction of sensed data that are not compatible with sensor nodes. For example, in An agriculture sensor node, the data transfer unit is the whole group of sensed data like pH level, light intensity, soil nutrient levels and much more. Similarly, for sensed data storage, record as a unit is a wise solution rather than a file. The record-based storage system is highly convenient over a typical file-based storage system for sensor nodes. The record-based storage system requires less operations for accessing data, easy data transfer and data tracking, and low computation power. Nowadays, microcontrollers have NVRAM, EEPROM as on-chip memories. These can also be used as storage for sensed data. This new framework aims to provide an optimized, low memory overhead storage system for a given set of sensors. This will help IoT/WSN application developers to easily work with the EEPROM/NVRAM like byte-addressable memories with multiple programming language support.

Contribution:

- The proposed framework provides a record as an abstraction for the sensed data.
- The proposed framework generates a library that includes the best-suited record structure for the selected sensors.
- The proposed framework generates a lightweight library in different languages such as C/C++/Arduino language as per user input.
- The proposed Framework provides an efficient and lightweight library for selected various user inputs like the size of EEPROM, list of sensors and, the programming language for a library.
- Implementation of the proposed framework proves that operations related to the sensor data become much lightweight in terms of computation, and memory utilisation becomes highly efficient.

As discussed earlier, for a particular WSN application, each sensor node collects a similar type of data to transfer, and the base station receives data. Hence, the proposed storage system works well for the sensor nodes and the base stations.

II. RELATED WORK

In this section, we survey some well-known storage systems based on log structure, record-oriented, and file-based for sensor nodes in WSN. The matchbox [16] is an integral part of the TinyOS operating system and provides a filing system for the sensor nodes. It has some novel features such as corrupted data detection, intact metadata and limited data loss. Those are being written in failures such as power off, low resource consumption, and atomic operations such as file renaming and the file deletion. The matchbox also handles numerous restrictions of the FLASH memories such as lower endurance, and a limited block erase. The matchbox system suffers from multiple vulnerabilities such as it allows only sequential read and appending write operations. We can not open the same file for reading and writing purpose in parallel. It deletes a file if renaming a file A over existing file B and the file B is not open. Its design meets essential requirements of the WSN such as WSN does not require multiple open files, security in any form, a hierarchical filing system, random file access, and the more than one readers or writers for the same file.

In 2004, Dai et al. [17] proposed Efficient log-structured flash file system (ELF) that is a log-structured flash-based file system for the sensor nodes. Its design is based on data types available for the storage and provides functionality based operations on it. ELF has conceptually divided data into the sensor data (require frequent read/write), configuration data (require fewer updates), and binary program image (require periodic updations). The ELF uses RAM, FLASH, and EEPROM to store different data structures. It uses EEPROM to store crash recovery metadata and other data structures. The ELF exposes functions such as open, create, rename, delete, append, and truncate. The proposed system also uses a similar log-structured system but in the absence of the file-based systems.

In 2008, Escobar et al. designed Sensor Node File System (SENFIS) [18] that is a storage management system with file-based operations for the sensor nodes. It also extends its support to the TinyOS operating system. It has two components storage management and a file system. In storage management, memory is divided into the logical partitions to provide simultaneous access of the multiple file systems. The SENFIS contains a partition table and also exports partition table management primitives. The file system of SENFIS is a character-oriented UNIX based file system. These characteristics of SENFIS provides a high level of abstraction for actual read/write operations performed in underlying WSN OS such as TinyOS. In SENFIS, the FLASH memory is divided into the sectors, blocks, and the pages. Each sector is further divided into the pages with each page size of 264 bytes (actual 256 bytes of data and 8 bytes of metadata).

EEPROM File System (EEFS) [19] is a custom EEPROM file system developed by NASA for the data storage purposes. It is slot based file system where each slot has a fixed

TABLE I
Motes/Sensor nodes Comparison [12], [13], [14]

Mote/Sensor Node	Processor	SRAM	Flash	EEPROM	OS Support
MICA	ATmega128L	4KB	128KB	4KB	TinyOS
WisMote Mini	Atmel AVR 8 bit	16KB	128KB	4KB	Contiki
WisMote RS	Atmel AVR 8 bit	16KB	128KB	4KB	Contiki
Arduino BT	ATmega328P	2KB	32KB	1KB	-

TABLE II
Memory Technology Comparison [15]

Parameters	EEPROM	FLASH	SRAM	NVSRAM	EERAM
Density	128bit-4Mb	512Kb-64Mb	64Kb-1Mb	512Kb - 1Mb	4Kb-1Mb
Read Times	Medium	Medium	Medium	Medium	Medium
Write Times	Medium	Medium	Instant	Instant	Instant
Data retentions	200+ Years	100+ Years	Volatile	20+ Years (With battery)	100+ Years
Typ. Standby Current	1 μ A	15 μ A	4 μ A	4 μ A	4 μ A

contiguous region inside the memory. Each slot in the memory has a file header and file data. In the EEFS system, each slot can be of different sizes. The EEFS system has a pre-defined file allocation table in each slot and maximum slot size. This file allocation table is a fixed size regardless of the files in the file system. It also has an inode table that is similar to the FAT, but it is in RAM, and that contains the physical address of each file rather than the relative offset as it is used in FAT. The notable feature of EEFS is its straightforward implementation and the lower memory overhead.

In 2016, Penson et al. presented Tiny Embedded File System (TEFS) [20] that is a file system which uses a simplified tree indexing structure for the fast random reads and writes over the FLASH storage. The TEFS guarantees that the number of page reads and writes remains constant with the increase of file size during random read-write operations. It has some novel features such as better performance, a smaller code footprint, and a lower RAM consumption than the existing industry-standard file system available for the resource-constrained devices. The TEFS provides functions such as reading and writing records to and from the pages with opening, closing, and removing a file. In TEFS, the first section contains information about the file system and the cluster state section follows that. The cluster state section includes a bit field that provides the cluster's status either in use or free. The third section contains a single root directory with a fixed size initialized during formatting. In TEFS, each file contains a directory entry that includes metadata of the file. The TEFS also represents index structures as a tree rather than a linked list of the entries used in File Allocation Table (FAT). The sequential read and write times for TEFS are 10 to 20% faster than the FAT.

Recently, in 2020, Aras et al. put forward Micro Vault [21] that is a software-based reliable storage unit for the resource-constrained IoT devices. It mainly focuses on the lifetime of IoT devices such as FLASH memory endurance. It achieves this by adopting techniques such as grey coding, error-correction, micro-caching, and wear-levelling. The micro vault API uses various parameters such as wear-levelling size, read/write rate per day, and the data consistency requirements. Based upon that, it establishes a lower bound on guaranteed lifetime. The micro vault is written in C language, and it

supports AVR, STM32, and MSP430 MCU architectures. The micro vault is also compatible with many embedded operating systems such as FreeRTOS or the TinyOS. It also supports frameworks such as Arduino or PlatformIO.

These listed storage systems are file-based. In this paper, we present a novel storage system based on the type of data to be stored. The sensor nodes in WSN do not require file-based storage, but a record-oriented storage system is also preferable approach. The record of the sensed data on a sensor node is the fundamental idea behind the proposed method.

III. DESIGN CONSIDERATIONS

At the early phase of WSN design, each sensor node had assigned a fixed type of sensors based on application. Hence, for a particular application, the sensor node transmits the similar type of data. Therefore, it is a good idea to combine the sensed data from different sensors into a single record rather than making a separate file for each sensor data. Secondly, in case of connectivity failure, we need a lightweight storage system that helps us to achieve the zero percent data loss and allows us to store data and transmit those data later on.

A. Representation of sensed data

As each sensor node has connected sensors, we can design a single record for data storage. Examples of different application based on WSN are given below.

- 1) **Healthcare monitoring** : In a typical healthcare monitoring applications, each sensor node is connected with sensors such as pressure sensor, ECG sensor, EEG sensor, temperature sensor, flow sensor, accelerometer, biosensor, and encoders [22] as shown in Figure 2.
- 2) **Smart Agriculture** : In any smart agriculture system, each sensor node receives data from location sensor, optical sensor, pH sensor, humidity sensor, temperature sensor, and airflow sensor [24] [25].
- 3) **Industrial Environments** : We need sensors for temperature monitoring, vibration detection, flow rate analysis, and strain monitoring for developing any smart industry [26].

Figure 2. Smart Health Monitoring [23]

B. Variety of records based on WSN applications

As discussed in the previous section, a single record of sensed data is valid for the specific application of WSN. But, that same record may not be accurate and suitable for other applications. For example, the format of record from the sensor node in the smart agriculture system will be different then the healthcare system due to variation in types of sensors.

C. Connectivity Issue

In WSN, we connect the sensor nodes with base station or gateway using various technologies [27] [28] such as WiFi [29], ZigBee [30], and Bluetooth [31]. These technologies depend on power consumption, and sometimes it may not have the necessary power to transmit the received data. That's why the problem of connectivity may arise in these technologies [27]. In that case, we requires a lightweight buffer for these sensed data.

D. Garbage Collection

Mostly in case of connectivity failure or busy base station, we store the records in the buffer. After the re-establishment of connectivity or free base station, records will be transmitted and acknowledged. Hence, it's good to overwrite this non-useful acknowledged data. So, in next write cycles, the memory can be re-utilized.

E. Wear leveling

Solid-state storage devices (SSD) such as FLASH and EEPROM are storage mediums in sensor nodes. These SSDs have limited numbers of write/erase cycles. Typically any byte (in EEPROM)/page (in FLASH) of a NOR type memory device is reliable up to 100 K to 1 million cycles, and after this, it will be out of use. Thus, it's essential to entertain each byte/page of memory equally. A technique called wear levelling [32] is used for even distribution of write/programming cycles, hence every byte/page equally entertained and thus increase the overall life of SSD.

IV. STORAGE SYSTEM DESIGN

The proposed system is designed to address design considerations discussed in the section III. This storage system can be implemented on sensor node having any kind of byte-addressable memory. Mostly EEPROM is available on sensor nodes; thus We present the proposed system for EEPROM. Sensed data and significantly less metadata stored into EEPROM. EEPROM can be internal or external. Sensors are connected to sensor node are specific to the environment where this system is to be deployed.

A sensor node that has EEPROM either internal or external, this proposed storage system can be deployed. The system can have any size of memory storage; the proposed system will modify it's parameters accordingly. The layout and view of the

Magic Number
Configuration Data
Date and Time
freeMemAdd
No. of Records
Read Pointer
Write Pointer
Record Size
Record 0
Record 1
:
Record 255
:

Figure 3. Layout of Storage system

storage system are shown in the Figure 3, apart from records, some metadata required to be stored in memory. Also, the size of the record and metadata is not fixed; hence it's not always possible to allocate all memory apart from metadata to the records. There may be a chunk of memory which is always less than a record size, that may be left unallocated due to memory fragmentation.

A. Framework Design

The framework is designed to facilitate end-user to quickly develop record structure and storage system API just by selecting valid sensors, memory size, and language in which API is required. It also provides an option for magic number and length of memory for user configuration data. The same framework is used for any typical WSN applications such as agriculture, Industry, Healthcare, and many more.

This UI provides a menu for various key parameters and based on it that will automatically generate record structure, system metadata, and API. The parameters are types of sensors that are used in an application, memory size, the magic number

for a storage system, length of memory for user configuration data, and language in which storage system API is required.

A host system is required to run this, and as an output, it provides API library which has API and a definition of record structure as per user selection. The configuration file

Figure 4. RecordBOX Framework Architecture

contains symbols and data specific to user input. The header file contains generic structures of sensor record and metadata. API file contains generic definitions of read, write, and other functions. As discussed in the previous section, the record is changed as per application. Hence, this framework allows us to auto generate a record for any WSN. Therefore it is providing easy and fast application development. This framework is providing a solution which is one for all.

B. Record Design

Record changes according to its application area because smart agriculture has some sensors while healthcare monitoring required another set of sensors. Using the proposed framework, end-user has not worried about this issue. Just by selecting sensors from the list, the record will automatically be generated. As an example, for memory selection, 4/8/16/32 KB of memory is given, and for sensors, there are temperature, gas, motor speed and colour given. The proposed system treats the timestamp as integral data of record, it will occupies 4 byte. Each sensor value has a fixed size of memory and type such as, the temperature sensor has a float value which occupies 4 bytes. Each time, sensor is selected according to its type and size that much memory must be allocated.

The framework has defined some configuration symbols that represent records according to user input. For the temperature and gas sensor, it generates the record of 12 bytes. Similarly, for all the sensors given in the example, it generates record of 20 bytes.

C. Records in Storage

The layout and logical view of the storage system are given in the Figure 3. Records are stored sequentially in memory. No metadata is required for each record. Read and write

operations for records are done through the metadata of the storage system. Memory for records is treated as a ring buffer to implement wear levelling technique [33], it is important to equally utilize all bytes/pages in memory, as discussed in subsection E of section III. Read pointer points to a memory location from where the record is fetched. Read operation is destructive means after reading operation that record may not be available for the next time, it may be overwritten by write operation in case of memory is full. Write pointer points to memory where the next write will take place. Records are stored in memory logically similar to the ring buffer.

D. Metadata

To track the records in memory, some metadata is required to be stored along with the actual sensed data. It is given in the Figure 3. Its size depends on which options are chosen by the user. Also, some configuration symbols are set for this metadata structure by the framework. It is similar to the super block of the file system.

The magic number is provided by the user using the proposed system's UI. It is the signature of the storage system and used by the application to validate the underlying storage system. Also, the size of user configuration data is provided using UI, and according to that, space is allocated in the metadata structure. The user uses this data for storing system-related parameters such as the unique ID of the sensor node, encryption key. Date and time are primary data for sensed data. It can be hardcoded or provided through the network. For the local system, this works as a reference.

FreeMemAdd is the memory address from where the memory for records starts. It is used by API to fetch/store a particular record. The other member of metadata store maximum number of records that memory can store. It is used by API to perform circular queue like operations. Next are, two pointers read and write. They point to the next read/write location. The last one is record size which is very useful because the record size depends on user input and it is not hardcoded; hence this will provide that value which is used in API to find out exact memory locations in memory.

Metadata structure's elements are fixed, but their size depends on the configuration selected by the end-user.

For example, if a total number of records is less or equal to 256 then the size of no records and both read and write pointers will be 1 byte, if it is greater than 256 but less than 64K, then the size of the above three elements will be 2 bytes. Similarly, other element's size changes according to user input.

V. STORAGE SYSTEM IMPLEMENTATION

The proposed work has both design of the storage system and a framework for easy development of such storage systems for various fields of WSN. Arduino Mega 2560 board is used for proposed storage system deployment and testing. This board has 8KB SRAM, 256KB FLASH, and 4 KB of EEPROM [34]. Selected options by application developer are 4KB of EEPROM, magic number 10, all listed sensors, and Arduino style API library as shown in Figure 5.

A. Framework Implementation

This proposed framework is designed to auto generates records, configuration symbols, and API based on user input. This framework is implemented on Ubuntu 16.04 with some magic of bash script, dialogue utility, bc, and awk utility. This can work on any Linux based system with the supports of the utilities as mentioned above. it's architecture is illustrated in Figure 4.

1) *UI Implementation*: To design UI, dialogue utility is used on the Linux platform. User needs to choose options according to the underlying architecture of sensor node. For the given hardware options are chosen as shown in Figure 5. All these configuration data are written into a configuration file. Also, some new configuration data are generated based on these data for further use.

2) *Configuration file*: After the first stage, UI completed, all configuration data stores into the configuration file. These include memory size, the magic number, size of configuration data, symbols for each sensor that are included, supported API language. Also, new data is derived from the former one. Based on memory size and record size, the number of maximum records value calculated, and also it's size is calculated. All these data is stored into a single configuration file.

3) *Header file*: This file includes generic structures of sensed data, a record and metadata. Based on the data in the configuration file, the perfect sized record and metadata structure generated after the compilation of source codes.

4) *API file*: API are available in multiple languages such as Arduino processing, C, and C++. As We are using EEPROM memory, all these lower-level API are included in these files as shown in Figure 6. These API files include all generic functions definitions. The Header file is included in these files, therefore according to the user input, proper function invoked.

5) *Library*: As a final output, a library is generated, which has API and structure definitions according to user input. This library can be included in Arduino IDE or linked with application codes directly. So, this framework provides easy development of storage systems for byte-addressable memory for sensor nodes in WSN. It accelerated application development in the area of WSN.

B. API implementation

Figure 7 presents the API exposed by this system to the application developer. All are generic functions, and according to configuration symbols, specific code will be executed. Some functions such as `Init_Mem()`, `ack()`, `Update_DayTime()`, `Read_Record()` and `Write_Record()` are directly used by application developer, while others are indirectly used by these API.

1) *Init_Mem() function*: First `Init_Mem()` function is called to set up the storage system. It will set metadata elements such as a magic number, user configuration data, date and time, `freeMemAdd`, number of records, pointers, and record size. After it calls the `Write_Metadata()` function and that function writes this structure into EEPROM. In metadata structure other

than `freeMemAdd` element are set directly using configuration symbols, but `freeMemAdd` requires calculation as below,

$$\begin{aligned} \text{FreeMemAdd} = & \text{sizeof}(\text{FreeMemAdd}) + \\ & 3 * \text{sizeof}(\text{no_of_records}) \\ & + \text{sizeof}(\text{record_size}) \end{aligned} \quad (1)$$

As a first step, this function must be called to set up the storage system; otherwise, without it, other API are useless.

2) *Read_Record() function*: This is used to read a record sequentially from memory, and it takes the first argument a pointer to record (`sensor_record*`) as defined in Figure ?? . It copies the data from storage into a given record pointed by the argument. The second argument (unsigned char `ack`) is about the destruction of current record. Once the record is transmitted or used, it is acknowledged thus, after that it's not good to keeps it in memory. Thus, this argument provides this luxury to application developers if they want destructive read or wait for an acknowledgement. If the second argument is 0 means, this record is not acknowledged yet then the read function will not destroy it. In this case, after acknowledgement of the record, the developer needs to separately call `ack()` function to update the read pointer. If it is one means this record surely transmitted or used; therefore, it is already acknowledged thus, no need to keep it after reading. In that case, it is destructive read function means once the record is retrieved, there is no way to retrieve it back. Record is there in memory, but for the system, it is destroyed hence at any point of time in the next `Write_Record()` function call it may overwrite. In the case where the read pointer is the same as the write pointer that means no new record is available to perform read operation, this function returns a negative value which means error. It also works as a garbage collector because, after destructive read, that record's memory is automatically available for the next write cycle. It uses `Read_Rp()` and `Updates_Ptrs()` to find out the record location in memory and update read pointer to the next record in case of the second argument is one means already acknowledged, respectively.

In Figure 8, working algorithm of the function is illustrated, and the formula of record location used in this function is given below.

$$\begin{aligned} \text{effectiveAddress} = & \text{FreeMemAdd} \\ & + (\text{record_size} * \text{read_pointer}) \end{aligned} \quad (2)$$

`ack()` function update the last read pointer value. This function must be called after the record is acknowledged, and that is before the next read function; otherwise, the same record will read by the this function.

3) *Write_Record() function*: This is used to write a record sequentially into memory. The argument of this function is the same as in the read function, a pointer to sensor record. It copies the sensed data from record to memory. It uses `Read_Wp()` and `Update_Ptrs()` functions to find out record location in memory and update the write pointer to the next record location, respectively.

In the case where write pointer and read pointer have the same value means no space for the new record, then this function returns a negative value which means error.

Figure 5. User Interface (UI) of RecordBOX

Figure 6. API architecture

```

int Init_Mem(unsigned char* config, t_DayTime* day);
int Read_Record(sensor_record* rec, unsigned char ack);
//0 for non-destructive read,
//1 for destructive read,

int ack(void);
int Write_Record(sensor_record* rec);
int Update_Ptrs(unsigned long ptr, unsigned char flag);
//0 for read and 1 for write

int Update_DayTime(t_DayTime* day);
int Read_Metadata(record_metadata* recmeta);
int Write_Metadata(record_metadata* recmeta);
unsigned long Read_Wp(void);
unsigned long Read_Rp(void);

```

Figure 7. Listing of API

The formula given below is used to find location of record to write.

$$effectiveAddress = FreeMemAdd + (record_size * write_pointer) \quad (3)$$

4) *Update_Ptrs()* function: It takes two arguments, first is a pointer to the location of the record and the second is a flag which is either 0 or 1. Flag is used to identify which pointer is needed to update either read or write. For flag value 1 write pointer is updated and for 0 read pointer is updated.

As discussed, records are stored sequentially, and logically it is a circular queue; thus, the respective pointer value is

Figure 8. Working algorithm of Read_Record function

incremented by one when this function is called. Also after an update, it checks if the pointer is pointing out of bound memory or not. If it is pointing out of boundary, then the respective pointer is reinitialized to 0.

5) *Update_DayTime()* function: This function is used to update local time on the system. It is used in scenarios such as system rebooting and synchronization between sensor nodes and base station.

6) *Read_Metadata()* and *Write_Metadata()* functions: To read and write metadata, these functions are used respectively. The application developers do not require to use these functions directly. They are using a pointer to metadata structure, as an argument (record_metadata*).

These functions are used in Init_Mem() function to initialize the proposed storage system.

7) *Read_Rp()* and *Read_Wp()* functions: Both these functions are used to read pointers for reading and writing,

TABLE III
Comparison of different storage systems [18]

Feature	ELF	Matchbox	LiteFS	CFS	RecordBOX [Proposed]
Data Abstraction	File	File	File	File	Record
OS/Software	TinyOS	TinyOS	LiteOS	Contiki	EEPROM driver
Sensor Platforms	Mica2	Mica Family	MicaZ	Tmote/ESB/AVR MCU	AVR MCUs
Memory Allocation	Dynamic	Static	Dynamic	Dynamic	Dynamic
Memory chips used	RAM,flash	Flash	RAM,EEPROM,flash,	-	EEPROM
Number of files opened	Unlimited	2 (Read/Write)	8	-	-
Meta-data size	14 bytes (per flash page) 14 bytes	8 bytes (per flash page)	8 bytes (per flash page) 168 bytes RAM	-	32 Bytes (Max)
	14 * (number of nodes)(RAM)		2080 bytes ROM	-	(for whole system)
Types of files	Sensor Data	Data files	Data	-	Sensor Data
	Configuration	Updatable	Binary applications	-	Configuration
	Data			-	Data
	Binary program Image	Reliable	Device Drivers	-	

respectively. As a return value, these functions return read pointer and write pointer values, respectively. These functions are used in Read_Record() and Write_Record() functions.

VI. EVALUATION

The proposed system implementation was evaluated for the performance of memory usage and read/write time. For listed all sizes of memory and all types of sensors, memory for metadata and unused memory are calculated. Read and write times for different sized single records are calculated on a given platform.

1) *Memory Overhead*: Metadata is also considered to be memory overhead, but it is mandatory for storage management. Apart from the metadata, the remaining memory is equally divided for records which may introduce a memory that is always less than a single record in size. It remains unused during the lifetime of the current storage system. For different possible record sizes, required memory overhead is given in Table IV. Memory overhead includes metadata and unused memory. Percentage of memory overhead is calculated with respect to total records size. For all given possible inputs, memory overhead is less than 1% of total records size.

2) *Key parameters comparison*: Current implementation of RecordBOX is compared with bare metal arduino style programming because, we are using RecordBOX library in arduino style. There are several key parameters needs to be considered like read/write time, memory overhead, techniques for memory utilization, and ease of application development. This comparison is given in Table V. Also, energy consumption for the write operation of byte and record is given in Table VI. User options are as per example, discussed in section V.

3) *Comparison with existing storage systems*: In this subsection, we compare the proposed "recordBOX" with the other existing storage systems. Table III provides detailed comparative analysis based on certain parameters such as software support, sensor platform, type of memory used, and it's allocation and supported data types. The proposed storage system

TABLE IV
Memory Utilization of RecordBOX

EEPROM Size (Bytes)	Total Records Size (Bytes)	Memory Overhead (Bytes)	Memory Overhead (%)
4096	4080	16	0.39
	4080	16	0.39
	4080	16	0.39
	4080	16	0.39
8192	8176	16	0.19
	8172	20	0.24
	8176	16	0.19
	8160	32	0.39
16384	16368	16	0.10
	16368	16	0.09
	16368	16	0.09
	16360	24	0.14
32768	32752	16	0.04
	32748	20	0.06
	32752	16	0.04
	32740	28	0.08

TABLE V
Comparison of EEPROM programming with arduino programming language

	Arduino programming	Arduino programming with RecordBOX
Max Write time/record(ms)	50	57
Max Read time/record(μ s)	28	32
Memory overhead	0%	0.39%
Wear leveling	No	Yes
garbage collection	No	Yes
Easy development	No	Yes
Unified approach	No	Yes

TABLE VI
Energy requirements in write RecordBOX operation

Data	Time	Energy
Byte	3.3 ms	65.34 μ J
Record	57 ms	1.12 mJ

can be implemented on any micro-controller/microprocessor with EEPROM memory and related software packages. The real strength and novelty of the proposed "recordBOX" lies in it's the dynamic and unified approach of system configuration and maximum of 0.39% memory overhead.

VII. CONCLUSION AND FUTURE SCOPE

In this paper, we have presented the design and implementation of the "RecordBOX", a fast, lightweight, and minimal footprint storage management system for sensing nodes. This system uses a different approach to store the sensed data and configuration data as a record rather than a file. This system provides a framework to quickly design a storage system based on user needs for WSN based application. It has a minimal memory overhead and straightforward API in various languages. The "RecordBOX" uses wear levelling and garbage collection techniques to increase the EEPROM memory lifetime. Also, it accelerates the development of the WSN based application, just by employing user inputs, the library will be ready for use in the user-specified language. It uses less than 1% of memory size for the metadata to manage the sensor's data in similar read/write time. The "RecordBOX" provides ease of development, support of wear levelling and garbage collection techniques with a unified approach for WSN based application.

We can further extend the "RecordBOX" to integrate into real-time operating systems such as TinyOS, Contiki, and LiteOS. We can also enhance the security aspects, and data recovery approaches. We can also extend the proposed "RecordBOX" for automatic sensors inclusion in framework.

REFERENCES

- [1] B. Carter and R. K. Ragade, "Message transformation services for wireless sensor networks (MTS-WSN)," in *Proceedings of the 2006 International Conference on Wireless Networks, ICWN 2006, Las Vegas, Nevada, USA, June 26-29, 2006*, H. R. Arabnia, Ed. CSREA Press, 2006, pp. 3–7.
- [2] T. de Almeida Oliveira and E. P. Godoy, "Zigbee wireless dynamic sensor networks: Feasibility analysis and implementation guide," *IEEE Sensors Journal*, vol. 16, no. 11, pp. 4614–4621, 2016.
- [3] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating systems for internet of things low-end devices: Analysis and benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 375–10 383, 2019.
- [4] X. Huan, K. S. Kim, S. Lee, E. G. Lim, and A. Marshall, "A beaconless asymmetric energy-efficient time synchronization scheme for resource-constrained multi-hop wireless sensor networks," *IEEE Transactions on Communications*, vol. 68, no. 3, pp. 1716–1730, 2020.
- [5] M. Amjad, M. Sharif, M. K. Afzal, and S. W. Kim, "Tinyos-new trends, comparative views, and supported sensing applications: A review," *IEEE Sensors Journal*, vol. 16, no. 9, pp. 2865–2889, 2016.
- [6] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The liteos operating system: Towards unix-like abstractions for wireless sensor networks," in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*. Ieee, 2008, pp. 233–244.
- [7] S. Aslam, M. P. Michaelides, and H. Herodotou, "Internet of ships: A survey on architectures, emerging applications, and challenges," *IEEE Internet of Things Journal*, 2020.
- [8] Y. Zikria, S. Kim, O. Hahm, M. Afzal, and M. Aalsalem, "Internet of things (iot) operating systems management: Opportunities, challenges, and solution," *Sensors (Basel, Switzerland)*, vol. 19, no. 8, 2019.
- [9] S. Bansal and D. Kumar, "Iot ecosystem: A survey on devices, gateways, operating systems, middleware and communication," *International Journal of Wireless Information Networks*, pp. 1–25, 2020.
- [10] *HigH-performance Wireless sensor netWork node*, Crossbow, 2014.
- [11] CDAC, "Ubiquitous computing," 2014. [Online]. Available: https://www.cdac.in/index.aspx?id=pe_u_project_details
- [12] *WIRELESS MEASUREMENT SYSTEM, MICA*, 1 2017. [Online]. Available: http://www.cmt-gmbh.de/Produkte/WirelessSensorNetworks/Datenblaetter/MOTE-KIT_MICA2_Datasheet.pdf
- [13] *WIRELESS MEASUREMENT SYSTEM, Arduino BT*, 2014. [Online]. Available: <https://www.arduino.cc/en/Main/ArduinoBoardBT?from=Main.ArduinoBoardBluetooth>
- [14] *8-bit Atmel Microcontroller with 128KBytes In-SystemProgrammable Flash*, Atmel, 6 2011.
- [15] Microchip, *World's Most Reliable Memory*, 2020 (accessed September 22, 2020). [Online]. Available: <https://www.microchip.com/design-centers/memory>
- [16] D. Gay, "Matchbox: A simple filing system for motes," <http://cs.uccs.edu/~cs526/mote/doc/matchbox-design.pdf>, 2003.
- [17] H. Dai, M. Neufeld, and R. Han, "Elf: An efficient log-structured flash file system for micro sensor nodes," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 176–187. [Online]. Available: <https://doi.org/10.1145/1031495.1031516>
- [18] S. Escobar, J. Carretero, F. Isaila, and S. Lama, "A lightweight storage system for sensor nodes," 2008.
- [19] NASA, "Eeprom file system," 2014. [Online]. Available: <https://github.com/nasa/eefs>
- [20] W. Penson, S. Fazackerley, and R. Lawrence, "Tefs: A flash file system for use on memory constrained devices," in *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2016, pp. 1–5.
- [21] E. Aras, M. Ammar, F. Yang, W. Joosen, and D. Hughes, "Microvault: Reliable storage unit for iot devices," in *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2020, pp. 132–140.
- [22] V. Vippalapalli and S. Ananthula, "Internet of things (iot) based smart health care system," in *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPEs)*, 2016, pp. 1229–1233.
- [23] A. R. Al-Ali, M. Al-Rousan, and M. Al-Shaikh, "Embedded system-based mobile patient monitoring device," in *16th IEEE Symposium Computer-Based Medical Systems, 2003. Proceedings.*, 2003, pp. 355–360.
- [24] Steven Schriber, "Smart Agriculture Sensors: Helping Small Farmers and Positively Impacting Global Issues, Too," <https://www.mouser.in/applications/smart-agriculture-sensors/>, 2013.
- [25] D. D. K. Rathinam, D. Surendran, A. Shilpa, A. S. Grace, and J. Sherin, "Modern agriculture using wireless sensor network (wsn)," in *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*, 2019, pp. 515–519.
- [26] Kay Soon Low, W. N. N. Win, and Meng Joo Er, "Wireless sensor networks for industrial environments," in *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, vol. 2, 2005, pp. 271–276.
- [27] M. Farsi, M. A. Elhosseini, M. Badawy, H. Arafat Ali, and H. Zain Eldin, "Deployment techniques in wireless sensor networks, coverage and connectivity: A survey," *IEEE Access*, vol. 7, pp. 28 940–28 954, 2019.
- [28] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of things (iot) communication protocols: Review," in *2017 8th International Conference on Information Technology (ICIT)*, 2017, pp. 685–690.
- [29] M. E. E. E. Alahi, N. Pereira-Ishak, S. C. Mukhopadhyay, and L. Burkitt, "An internet-of-things enabled smart sensing system for nitrate monitoring," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4409–4417, 2018.
- [30] M. S. Farooq, S. Riaz, A. Abid, K. Abid, and M. A. Naeem, "A survey on the role of iot in agriculture for the implementation of smart farming," *IEEE Access*, vol. 7, pp. 156 237–156 271, 2019.
- [31] Q. D. La, D. Nguyen-Nam, M. V. Ngo, H. T. Hoang, and T. Q. S. Quek, "Dense deployment of ble-based body area networks: A coexistence study," *IEEE Transactions on Green Communications and Networking*, vol. 2, no. 4, pp. 972–981, 2018.
- [32] S. K. Panigrahi, C. Maity, and A. Gupta, "A simple wear leveling algorithm for nor type solid storage device," *CSI Transactions on ICT*, vol. 2, pp. 65–76, 2014.
- [33] V. Turau and C. Weyer, "Long-term reliable data gathering using wireless sensor networks," in *2007 Fourth International Conference on Networked Sensing Systems*, 2007, pp. 252–259.
- [34] *8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash*, Atmel, 2 2014.

Yagnik Bhavsar Yagnik Bhavsar is TA in the SoT, PDPU, India. He is working in area of WSN, IoT, embedded software development, embedded system design, and robotics.

Chintan Patel Chintan Patel is a research scholar in the SoT at PDPU, India. He has published papers in various reputed journals such as WPC and IEEE TC. He is also an active member of the IEEE, ACM and CSI.