Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Efficient Wireless Incremental Updates to Resource-Constrained Devices

Colin McDonagh

M.A.I. (Computer Engineering)

Supervisor: Jonathan Dukes

Submitted to the University of Dublin, Trinity College, May, 2018

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request

Colin McDonagh | 9/5/18

Name | Date

Summary

Wirelessly updating resource-constrained devices - like nodes within a Wireless Sensor Network - is necessary to fix on-board firmware bugs and alter program functionality. Primarily due to RF transceiver usage, updating may consume significant energy resources of a device. However, this cost may be minimized by intelligently reducing update sizes, like through only sending a patch containing bytewise differences between current on-device firmware and new firmware; i.e. an incremental update.

Research shows that compressed incremental updates reduce energy usage significantly; especially those which use the *diff* algorithm BSDiff. In this thesis, a prototype implementation using said algorithm is carried out to encounter, document and overcome challenges of incrementally updating particularly memory limited resource-constrained devices.

Some problems encountered in incrementally updating a memory-constrained device consist of determining how to patch firmware without reducing the maximum allowed patch size, dealing with flash write minimum erasure units, and page alignment. The corresponding solutions synthesized to counter these challenges consist of patching firmware in-place over old firmware, using parts of the received patch to store otherwise overwritten and lost old firmware needed for the patching process, and altering the structure of patch files.

Altogether, these solutions allow for an incremental implementation that requires no additional memory other than that which stores old firmware and the received patch. Furthermore, due to transmitting smaller updates, and the negligible patch size increase due to reformatting, the incremental mechanism should tend to result in significant energy savings.

Acknowledgements

I would to thank Jonathan who proactively gave of his time throughout the year; especially during the trickiest stages of development. The constant support encouraged me to remain consistent in my development efforts and writing.

I would also like to thank Jeremy Jones for agreeing to act as second reader, and for providing his feedback on project work after the demonstration which helped to guide last efforts.

Table of Contents

List of Figures

List of Tables

List of Acronyms

WSN:	Wireless Sensor Network
BLE:	Bluetooth Low Energy
OTA:	Over-the-Air
OOM:	Out of Memory
RF:	Radio Frequency
SDK:	Software Development Kit
VM:	Virtual Machine
IP:	Internet Protocol
RAM:	Random Access Memory

1 Introduction

"The journey of a thousand miles begins with one step."

– Lao Tzu

1.1 Research Focus

This thesis aims to encounter, document and overcome challenges in designing and implementing incremental, diff-based update mechanisms for resource-constrained devices with limited energy supplies and memory. Furthermore, particular emphasis is given to the documenting of an implementation of such an incremental process in the context of both limited RAM, and just as importantly, if not more so, limited persistent storage. This decision is made primarily on the basis of the novelty of such research.

1.2 Background

Resource-constrained devices, including nodes within a Wireless Sensor Network (WSN), may be deployed to locations that are relatively or even extremely hard to access physically, depending on what their purpose is. For example, consider smart building lighting systems which only light rooms that are occupied. Here, motion sensing devices that sense people are often embedded within partition walls, especially in the context of a family home, thus making them difficult to access. Another example is use of discrete location aware devices which are camouflaged and attached to sparsely located trees in the Amazon rainforest to prevent illegal harvesting [1]. Considering that the Amazon rainforest is larger than the USA, these devices are not going to be easily reachable; if at all.

The first and most obvious challenge posed by remotely or awkwardly located resource-constrained devices is the gradual depletion of energy resources. As the devices are hard to maintain physically, renewing energy resources, through inserting new batteries into the device, for example, is also going to be an arduous task. Therefore, if energy resources may not be renewed easily, there is an increased importance placed on using less energy to maximize device longevity.

Tangential to the above, while these devices are deployed, bugs in device behavior may be encountered, whether they’re implementation related, design related, or even security related. However, regardless of their nature, even the smallest bug could compromise firmware drastically, especially in the context of mission critical software.

Furthermore, new versions of firmware may also be written which include program parameter reconfiguration or new feature designs. However, regardless of the specific reasons for writing new firmware, it’s apparent that redeploying these changes to devices situated in hard-to-access locations, like devices embedded under floors, through a physical connection, including USB, may be unrealistic, inefficient and costly.

Therefore, the second challenge in deploying resource-constrained devices is how to redeploy new changes efficiently without the need to physically connect to devices.

Primarily, Over-The-Air (OTA) wireless transmission protocols can be used as a solution. OTA protocols, including Bluetooth Low Energy (BLE), that allow for communication within a large range are especially useful. As updates may then be transmitted to resource-constrained devices wirelessly, the need to tear down infrastructure, or perform any other arduous tasks to access devices, is removed. Despite this, given the limited range of such protocols, the solution in itself doesn’t provide a means of accessing remotely located devices. However, IP addressable gateway devices, commonly connected to an AC line, may be used to forward firmware updates sent from a developer’s host device to resource-constrained devices within the proximity of the gateway, thus also removing the need to be on-premise to carry out updates.

In introducing the use of wireless communication into the process however, arguably the greatest concern is the energy consumption of updates [2]. On unconstrained devices, like PCs laptops, this isn’t a major concern: rather, the time to complete an update is more important. Although reprogramming duration is important to constrained-devices as well, because they may not be able to function normally during this period, the reasons for prioritising energy-use minimisation primarily constitute the fact that there are numerous potentially high-energy demand stages, relative devices’ resources, in an

OTA update. These include: wirelessly receiving the updates, processing them, and writing the new firmware to persistent flash storage. Even though updates may not occur often, it’s important that they do not deplete a considerable amount of resources. Furthermore, before an update is even triggered, it’s important to know if a device has the energy required to complete the process. Therefore, the energy consumption of updates must be considered, and potentially optimized.

As to related research, it’s clear that two of the primary contributors to energy consumption during an update are the use RF transceivers and erasures of flash memory [3, 4]. Fortunately, certain measures may be taken to reduce both of their usages. Beyond protocol related configuration, like using an optimal transmission rate, constrained-device RF transceiver usage may be reduced through reducing update sizes. Furthermore, since the client must also respond to the host with a success message for every update packet it receives, on the basis that guaranteed delivery is of vital importance [5], reducing update sizes also reduces the number of transmissions to the host, thus reducing energy consumption further.

Concerning flash memory erasures, this is also intrinsically related to update sizes, as the larger the new received firmware is, the more flash that must be erased and rewritten. Therefore, seemingly, decreasing update sizes is the primary means of minimizing update process energy consumption. However, reducing firmware sizes is not a viable option, as it’s restrictive, and wouldn’t necessarily result in large savings in any case. Therefore, alternative means of reducing update sizes should be sought after.

Compression is one option to reduce the size of updates. In theory, as CPU active time consumes approximately one tenth of the energy required by a RF transceiver on a typical device [6] compression should be a suitable means of reducing energy consumption. However, depending on the compressibility of an update, compression could also increase energy consumption [3], especially in the case of monolithic updates, which renders compression unhelpful for the time being.

Tangential to the above however, it’s interesting to note that a firmware update may not necessarily contain large amounts of changes compared to firmware currently executing on a device, which means that sending whole new firmware images may be rather inefficient or wasteful, especially in the case of minor bug fixes or parameter reconfiguration. Therefore, one solution to reducing update sizes is to send only parts of new firmware that have been changed in comparison to a previous version.

Using a change-based approach, updates could then constitute whole functions which have been partly altered, new modules of firmware, or even a line-by-line difference between executable versions such as would be generated by the Unix *diff* command; the last known as an incremental update.

Given these methods may reduce the size of the updates significantly, the focus is then on how to build such systems which allow change-based updates to be applied in the context of limited RAM and persistent storage, which will be of particular interest to embedded vendors. With specific interest given to incremental approaches, this thesis then sets out to do just so: to encounter, document and overcome challenges in designing diff-based update mechanisms for energy and memory constrained devices.

1.3 Personal Motivation

The reason for carrying out such research is to provide a sufficient dissertation for a MAI degree, specifically in computer engineering, at Trinity College, Dublin.

1.4 Thesis Structure

This thesis is composed of 6 chapters, 5 excluding the Introduction. Figure 1 displays an infographic of the chapters, which includes a question each chapter poses, and suggests an order in which they which may be read.

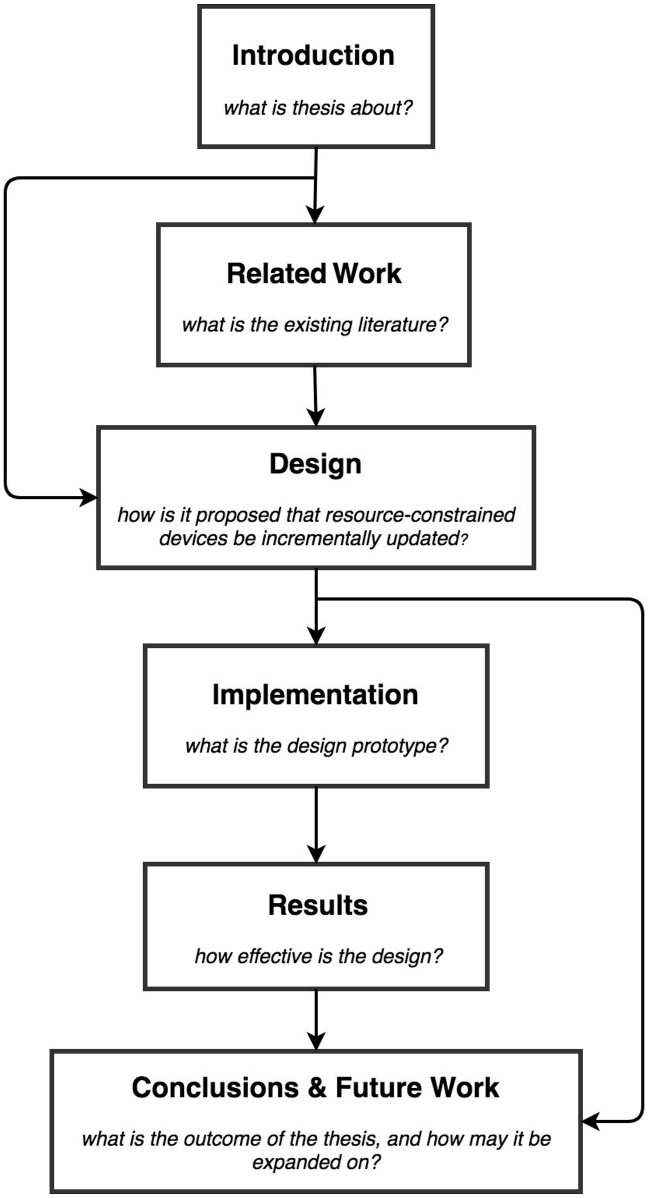
Introduction
what is thesis about?
Related Work
what is the existing literature?
Design
how is it proposed that resource-constrained devices be incrementally updated?
Implementation
what is the design prototype?
Results
how effective is the design?
Conclusions & Future Work
what is the outcome of the thesis, and how may it be expanded on?

Figure 1: Chapter infographic.

A brief description of the contents of each chapter is as follows:

- *Related Work*: relevant literature reviews for this thesis constitute work that's been carried out on update designs, including papers on unconstrained device and non-incremental mechanisms, but focuses on incremental research.

- *Design*: Outlines specific challenges in designing a resource-constrained incremental update mechanism, especially with regards to memory-constraints, and illustrates how they are overcome within this thesis.
- *Implementation:* Describes a working prototype of the update design, and the work carried out in doing so with respect to each significant part of the implementation process.
- *Results:* Details the appropriate findings from numerous update experiments.
- *Conclusions & Future Work:* Defines the overall outcomes of the project and describes a number of areas of work which would be beneficial to the implementation.

2 Related Work

“People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on eachother, like a wall of mini stones..”

– Donald Knuth

2.1 Criteria

The approach taken to reviewing material was a top-down approach: first reading literature related to high-level update approaches indiscriminate of the type of device being updated, and then focusing on change-based update mechanisms, mostly in the context of updating resource-constrained devices.

Different sources of relevant literature reviewed are included on the basis that they either:

1. Contribute to conceptualizing different software update mechanisms.
2. Enable a comparison between transparent and platform dependent update mechanisms.
3. Illustrate an incremental mechanism or detail a binary patching algorithm used in same.

2.2 Non-Transparent Updates

The most widely discussed means of updating in the context of unconstrained devices is by far dynamic updating: that is, updating a program without stopping its operation. Dynamic updates may take place in distributed systems [7] or cloud computing clusters, however, their use in updating single programs is of more interest: especially with respect to updating operating systems [8] or otherwise componentized software [9]. To perform a dynamic update, it’s necessary that at some stage, each part of the software is available to be rewritten, and thus, the greatest challenge is in identifying points at which modules may be updated.

However, since dynamically updating specifically relates to minimizing program downtime, solving the aforementioned update point problem in the context of resource-constrained devices is not necessarily a priority. However, using componentised firmware systems, that form the basis for dynamic updates [10], is of interest, as modular updates may reduce update sizes significantly through sending only the software modules that have changed between two versions of firmware. Especially in the case of transmitting minor bug fixes, these systems perform very efficiently, as such updates should only affect a small number of components, if not just one [11].

Furthermore, as components may be loaded onto separate parts of memory on a device, modular firmware may also help reduce the amount of update flash writes: since only firmware components changed need to be erased and rewritten. The presence of ‘slop spaces’, or unused memory, placed between components also helps reduce any necessary component reallocation due to increases in component sizes.

Example resource-constrained targeted modular systems include VM-like systems like Dynamic TinyOS [11], which use a bytecode interpreter, like Maté, to disseminate VM application code instead of whole firmware versions [12]. OS-like systems may support modular updating as well, including Contiki [13], however, this may pertain to updating parts of the operating system more so than application firmware run within the context of the OS.

Despite the energy savings in using modular updates, there are also two distinct disadvantages to using componentized firmware. Firstly, if updates are poorly managed, changes may span across numerous modules. The implications of such are each module would need to be re-sent: effectively carrying out regular a monolithic update. To solve this problem, developing could take place on only a small numbers of modules, however, updates may necessarily involve varied parts of a program.

Secondly, and more importantly, in order to update firmware modularly, the firmware itself must be written in a component-like manner. Another way of saying this is the firmware is not transparent to the update system: an arbitrary program executable on a constrained-device not componentized in the manner required by the device's modular management middleware will not be executable. Apart from the added design complexity introduced by requiring firmware to 'wire' together the different software components, the update system will also be very much platform-dependent. Therefore, this may be the strongest argument for the optimization of naive update mechanisms as opposed to non-transparent approaches: platform independency.

2.3 Incremental Updates

As opposed to sending components which have changed between two firmware versions, as takes place in a modular update, incremental updates send the bytewise difference between two firmware images. This requires using a binary delta generation library, somewhat like the Unix *diff* command, to represent these changes. When the delta-composed update is received by the device, it is then used so as to patch the intended firmware together from the old firmware already stored on the device.

The benefits to using an incremental or progressive approach to updating resource-constrained devices as defined above include:

- High efficiency, especially with regards to bug fixes and parameter reconfiguration.

- Patches have a highly compressible structure, so compression will be unlikely to not result in energy savings [3].
- The relative ease of developing platform-independent transparent firmware.

In terms of binary delta libraries, some update designs include altered versions of common algorithms, including *edit script* [14], which extends an algorithm like UNIX *diff* to allow rewriting parts of single firmware instructions through a *repair* opcode. Other approaches include using an optimized versions of algorithms, like Rsync [15]. However, depending on the degree of optimization, the underlying algorithm used may still have a limiting effect. For example, in Zephyr's optimized RDiff [16], block-level granularity is still used to determine transparent firmware changes, making for difficult management of cases where whole blocks must be sent when even one byte within each is changed [3]. Therefore, the base-algorithm used is important.

VCDiff and BSDiff, two other algorithms, have been shown to outperform traditional delta libraries in terms of generated patch file sizes, memory footprint and resource-constrained device processing requirements [3]. Furthermore, these results are supported by another set [4] in the case of the latter algorithm.

In particular, BSDiff is a novel algorithm [17], insofar as it doesn't use a “COPY” instruction to represent blocks of unaltered bytes in two firmware versions, but instead uses an "ADD" instruction to record extended blocks of unaltered bytes which could contain approximate matches, where at least 50% of the bytes must match in the extended regions.

One of the primary reasons that this method performs so well is because of the nature of changes to executables. In regions of firmware not directly affected by a modification, changes are sparse and modified addresses are likely only to change in their least significant one or two bytes [17]. Consequently, small changes are likely to be considered part of approximate matches. Therefore, as unaltered bytes will correspond to long sequences of zeros, and approximate matches will contain a bytewise difference in which at least half of the bytes are zero, the patch structure is extremely compressible. Especially when used with a Lempel Ziv form of compression, the algorithm has been shown to result in significant reductions in energy usage across various firmware change test cases [3].

Alas, as of the above, there has been, seemingly, a considerable amount of research carried out on the energy efficiency of incremental mechanisms and their delta generation algorithms. However, there are still numerous challenges to using an incremental approach in a resource-constrained context. Primarily, these constitute implementing incremental designs in highly memory-constrained environments. As opposed to with monolithic or componentized systems, incremental updates need to build the intended firmware from the old firmware, requiring a buffer of RAM to do so. If there isn’t enough RAM to wholly patch the new firmware together, then persistent-storage must be used instead. Furthermore, complications around maximum update sizes and overwriting needed parts of old firmware also ensue.

With respect to the aforementioned problems, there is an apparent lack of clear solutions provided through incremental update research papers. Furthermore, this is seemingly an area in which research would be somewhat novel, and beneficial, and therefore is the focus of this thesis.

3 Design

"There is no subject so old that something new cannot be said about it."

– Fyodor Dostoevsky

The primary methodology behind this research is the implementation of a working prototype to thoroughly encounter, document and overcome challenges in implementing a process of applying incremental updates to resource-constrained devices. Prototyping - one of the primary ways in which CS research is carried out - validates the feasibility of a design solution, and involves a gradual refinement of initial design ideas to produce more thorough and correct solutions, as well as a performance evaluation after the implementation has been finished [18].

In the context of this thesis, these initial design decisions include choosing to build on top of existing firmware which allows for carrying out regular monolithic updates, extending the software appropriately, and using *BSDiff* as the binary delta algorithm on account of its novelty and performance within resource-constrained devices; as outlined in Related Work. Furthermore, in order to better focus on design elements directly related to the patching process, the decompression of updates on resource-constrained devices is not considered.

Despite these initial decisions, other design is made in response to encountering first-hand the challenges in developing for a highly memory-constrained device, such as the issue of reduced maximum update sizes explored in section 3.1. To address one particular idea, some background into BSDiff patch files is necessary, and so is discussed in section 3.2. Furthermore, the restrictions inherent in the use of flash persistent-storage are discussed in section 3.3 to address possible reformatting of the patch file structure. Finally, an overview of the incremental process is given in 3.4.

3.1 Overwriting Problem

The first step to the incremental process is to generate a patch file for a device’s firmware on a PC and transmit the patch update to the resource-constrained device wirelessly.

Once the board receives the update, it begins to patch the new firmware together. As the RAM available may be limited, patching may have to take place on a page-by-page basis. The page of new firmware then needs to be written back into persistent storage, but a conceptual problem quickly arises: where does this page get written to?

If each page of new firmware is written to a new location within memory, not overwriting any old firmware, then this restricts the maximum update size significantly. This is because, if a set amount of persistent-memory is allocated to the update process, then this combined allocation will have to accommodate for the old firmware, the patch, and the new firmware simultaneously. This is displayed in Figure 2.

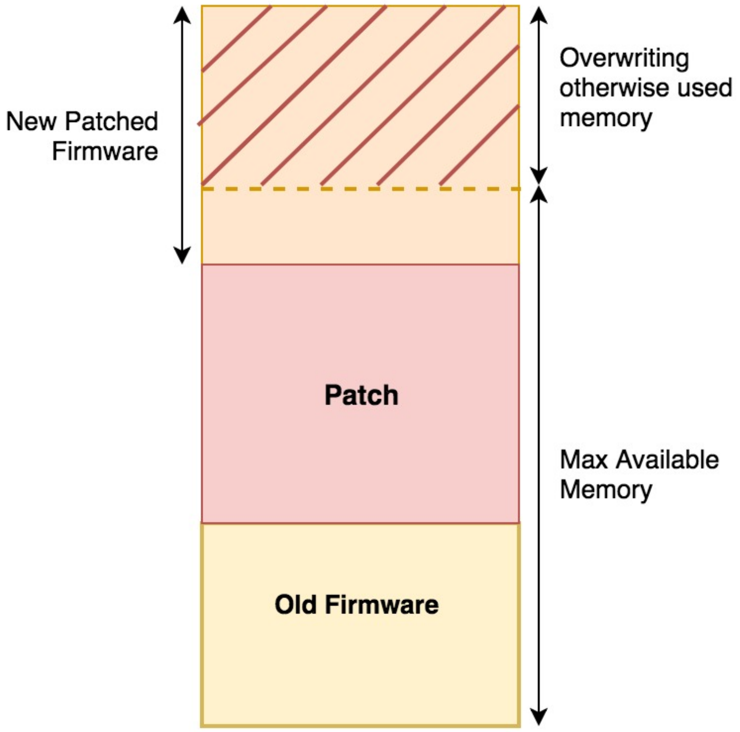

Figure 2: Limited flash memory available.

As update sizes increase, and patch sizes accordingly, the maximum update size is also further reduced. Therefore, this is not a desirable solution for highly memory-constrained devices, as the implication may be that realistically sized patches cannot be used.

Another idea is to write the new pages of firmware into where the old firmware is stored; applying the patch in place. However, this could jeopardize the patching process, since the new firmware is potentially constructed from parts of old firmware, which would be erased hence. The is illustrated in Figure 3.

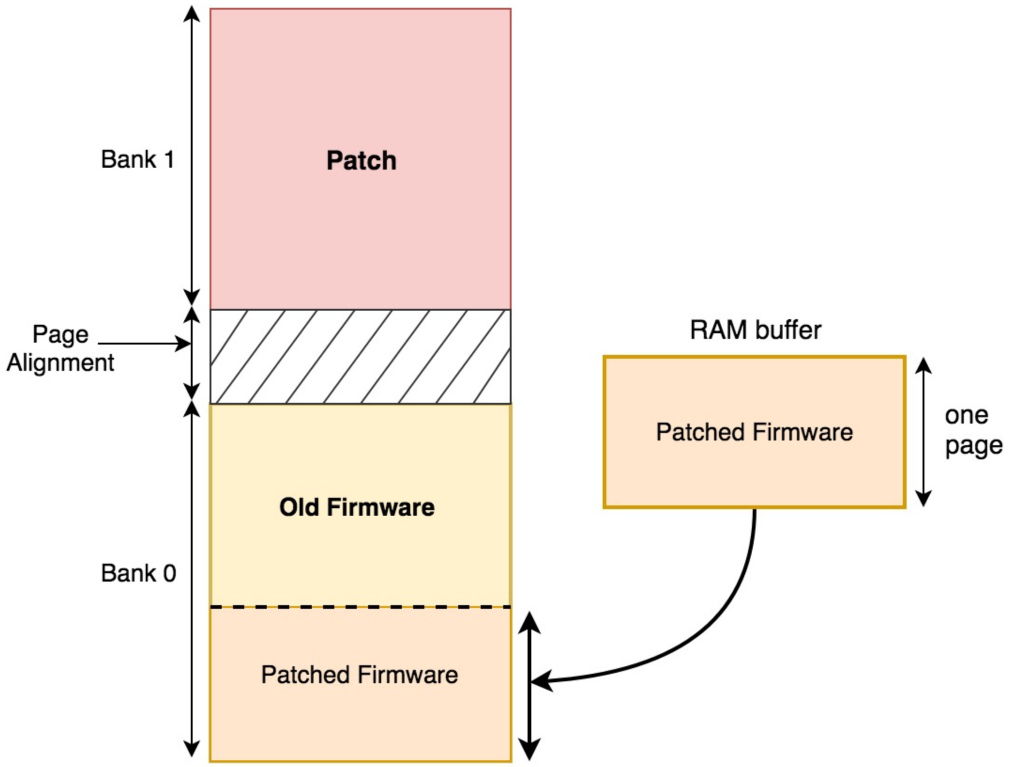

Figure 3: Overwriting patch problem. Page alignment padding may be used to write the patch starting from a new page boundary.

Regardless of the binary delta algorithm used in the process, once it consists of “INSERT” and “COPY”instructions, or an "ADD" instruction in the case of BSDiff, the above scenario is always a possible outcome, as one only must consider a simple case in which the first new firmware page consists entirely of “INSERT” bytes and the second new firmware page depends on the first page of old firmware, which has just been overwritten and lost.

One may also propose the firmware be patched in a non-sequential manner, such that new firmware page writes do not overwrite any parts of the old firmware which are needed to patch a future part of the new firmware. However, this could lead to Out of Memory (OOM) faults:

1. Flash memory, discussed at a later point in this chapter, dictates that the size of each of these non-sequential new firmware writes must be one page long.
2. It can’t be guaranteed that at least one contiguous page of old firmware isn’t used to patch new firmware.
3. Therefore, potentially subsets of pages of old firmware would need to be buffered in RAM during the process: this could be pages of RAM. Given, that one page may be all the available RAM, this approach is not appealing.

Therefore, simply overwriting old firmware with pages of new firmware, without first saving the appropriate old firmware pages, is not an acceptable solution.

In terms of places to save these old firmware pages, the final idea is to overwrite parts of the patch which have already been used to generate new firmware. However, to determine if this could potentially work, the makeup of patch files in BSDiff needs to be investigated.

3.2 BSDiff Patching

As mentioned previously in Related Work, BSDiff uses an "ADD" instruction and an "INSERT" instruction to represent differences between firmware. Use of “ADD” instructions is the novel aspect of the algorithm, and encodes blocks of new firmware as references to blocks of old firmware with bytewise differences which need to be added to the old blocks: for instance, exactly matching regions have a bytewise difference of zero.

Considering only this behaviour, the “ADD” instruction wouldn’t seem to offer any advantage over using a “COPY” instruction. However, the benefit to using “ADD” instructions is regions of approximately matching bytes before and/or after exactly matching regions may be represented by a bytewise difference. To better define the exact behaviour of the algorithm, the main steps to generating patches are:

1. Index the old file based on a suffix sorting algorithm [19].
2. Using this index, find regions that match exactly between the two firmware versions.
3. Generate a pairwise disjoint set of extended matches by considering the similarity of bytes before and after exact matches [17]. If the extended bytes match in greater than 50% of the cases, the exact matches are extended, and the difference between the old bytes and the new bytes within these extended matches are recorded as ‘ADD’ instructions.

4. Any bytes that are not covered by these extended matches are then covered by ‘INSERT’ instructions.

As approximate matches will consist of long sequences of zeros for exactly matching subregions, and the extended regions will be represented by zeros in at least 50% of the bytes, patch files are highly compressible; more so than copy and insert style algorithms.

Accordingly, a patch file’s content primarily consists of a “Diff Block” containing “ADD” instruction bytewise differences, and an “Extra Block” containing “INSERT” instruction bytes. This is illustrated in Figure 4.

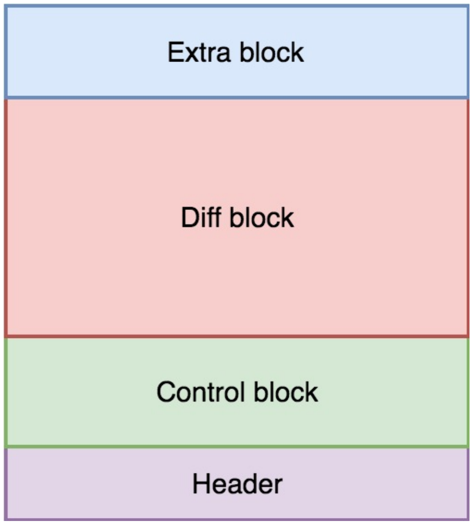

Figure 4. BSDiff patch format. The control block consists of commands to generate the new firmware from the diff block and control block, whereas the header contains pointers into the file.

As the size of the diff block is the same as the size of old firmware being added to, the patch's main content, the diff block and extra block, together equal the size of the new firmware. The implication of this analysis in the context of writing new firmware page-by-page is important, as it is: for each page of new firmware patched using the patch file and old firmware, there is always an equivalent page of used patch content which may be overwritten. This is illustrated in figure 5.

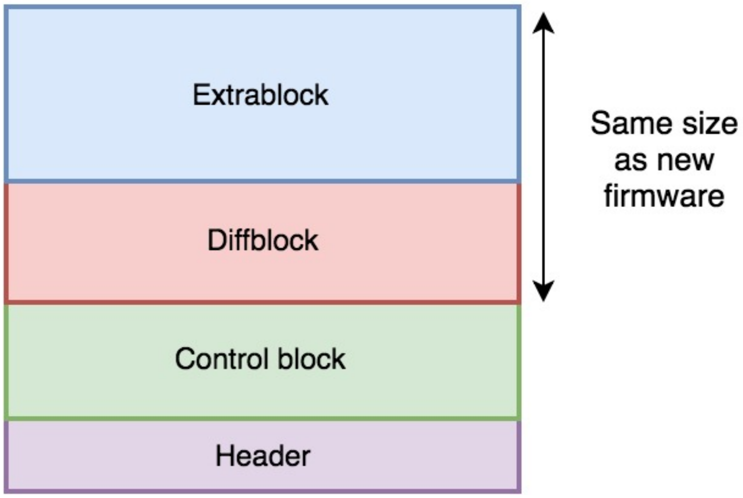

Figure 5. Firmware size is same as data block size.

Despite the above proof however, on the basis that the page of new firmware may have been formed from part of both the diff block and the extra block, will the non-contiguous nature of the free page affect its usefulness as a means of storing old firmware? Or can the page of old firmware be saved in part to both the diff block and extra block?

To answer these questions, the constraints of flash-memory usage must be first considered.

3.3 Flash Memory Restrictions

Considering the question posed in the previous section: before part of an old firmware page may be written in flash, it first must be erased. Furthermore, flash doesn't allow for the erasure of individual bytes: instead, a whole unit, which constitute a page, must be erased all at once. Therefore, it is impossible to save old firmware back into a patch file - one part within the diff block; the other within the extra block - without erasing parts of the patch which will be used to generate future pages of new firmware. This is illustrated in Figure 6.

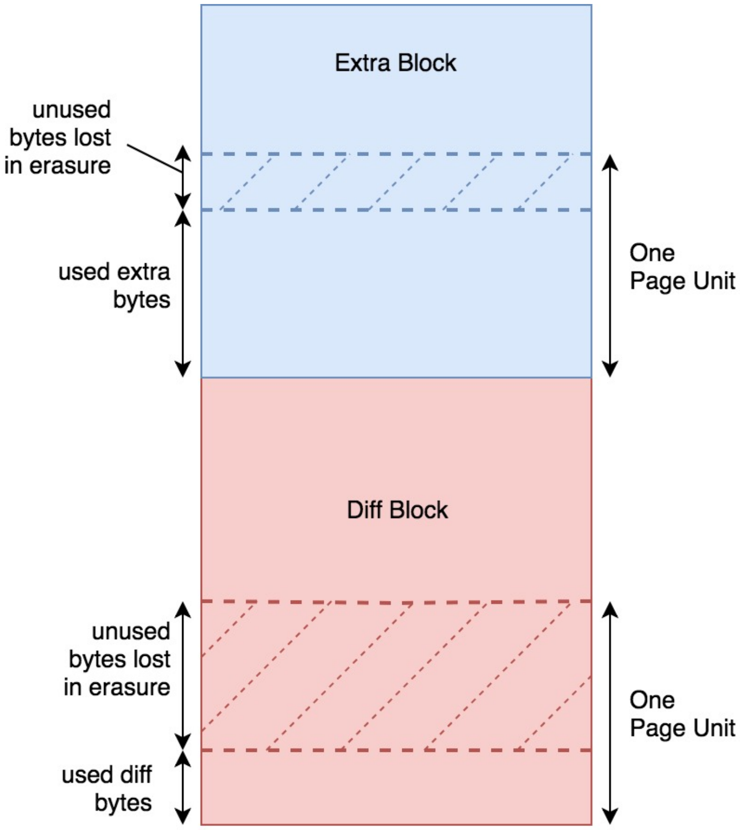

Figure 6: Saving split page issue.

However, as BSDiff stores bytewise additions and new “INSERT” bytes into these two separate blocks not as a mandatory step in the patch generation, but as a means of slightly increasing the compressibility of the patch, these two blocks may be merged. The alternative structure then consists of diff bytes and extra bytes ordered in the manner which they are used to form new firmware, which will now be referred to as the “Data block”. This is illustrated in Figure 7.

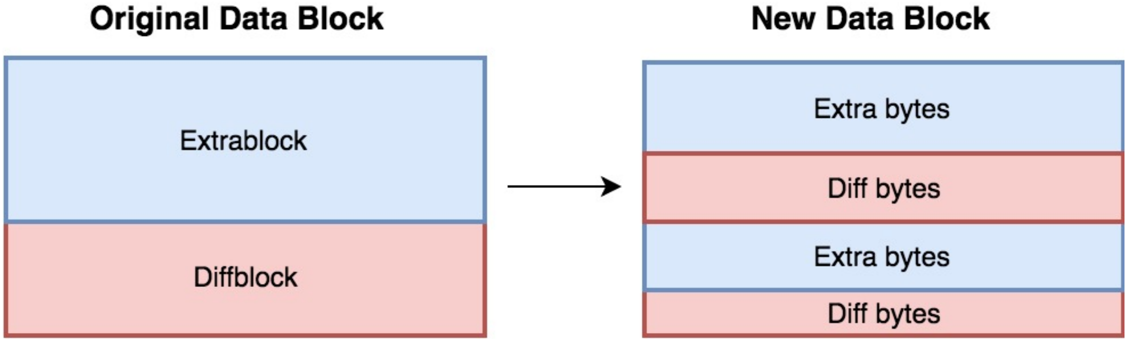

Figure 7: New data block format.

The above is therefore a solution to the previous non-contiguous nature in which patch content bytes were ordered. However, one other issue is of relevance.

Considering that the control block is made up of a variable number of control instructions, the offset of the data block within the patch file will likely not be one page. Furthermore, considering that the patch is likely to be stored starting from a new page boundary, this means the data block will also not be page aligned.

The above is a problem as each free page of patch will then traverse a page boundary, meaning that to write to overwrite this free page, one page on either side of the boundary will first need to be erased; losing unused parts of the data block and control block. This is illustrated in Figure 8.

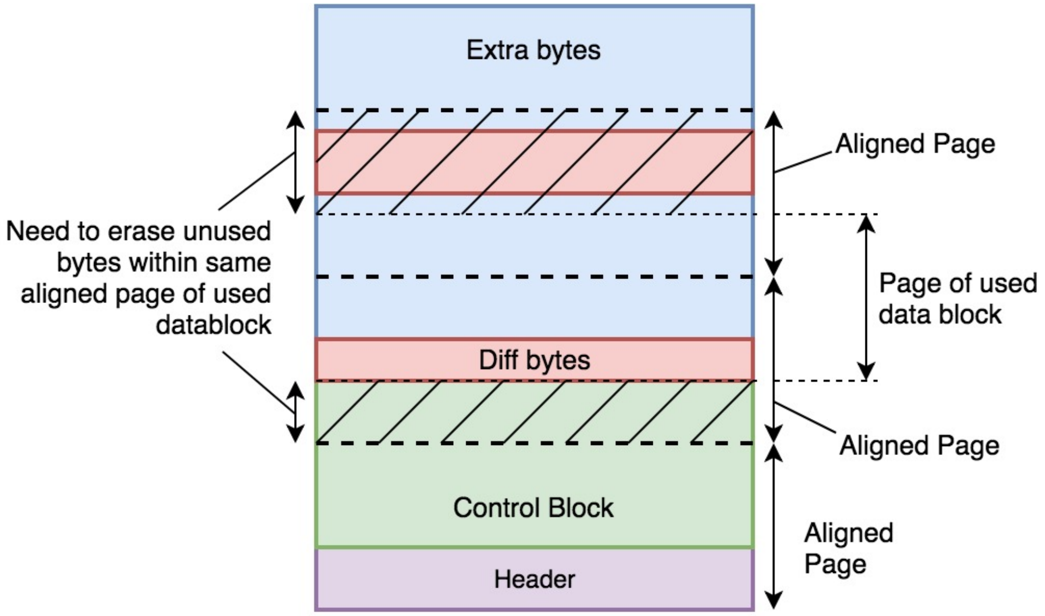

Figure 8: Losing unused bytes due to page alignment.

The solution to this problem is simple: move the header and control block to the top of the patch, such that the data block is at the start of the patch; which is page aligned. This is illustrated in Figure 9.

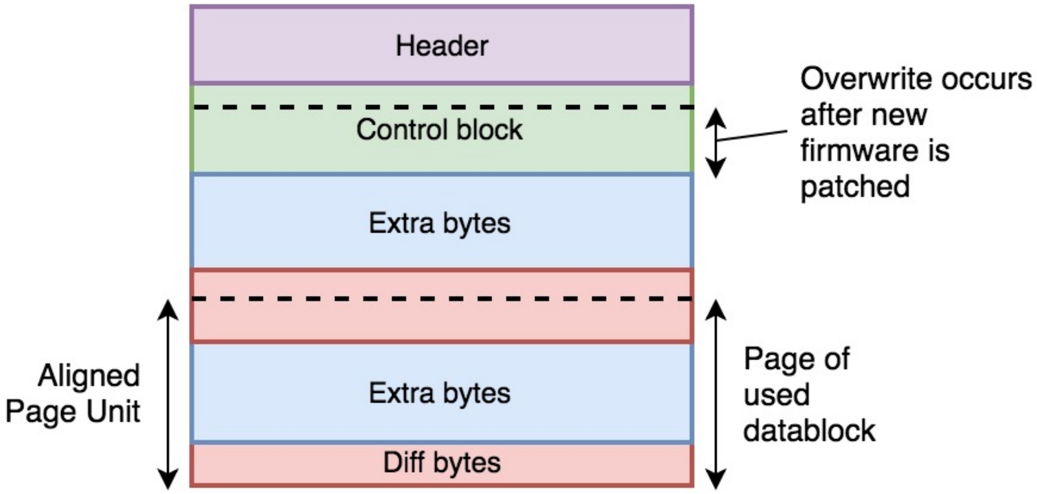

Figure 9: Page-aligned data block solution.

This rearrangement of the patch works even though the last overwrite in saving old firmware may erase a variable amount of the control block and even header; if the size of new firmware is not a multiplicative factor of page size. This is because the control block and header are no longer needed by the time they may be overwritten, since the last page patched firmware is stored in RAM; waiting to be written to flash.

Furthermore, it’s necessary to keep the header at the top, and not the control block, as the control block is of variable size, and determining the header’s offset would therefore be impossible.

With these modifications, the new patch structure looks like the following in comparison to the original patch, illustrated in Figure 10.

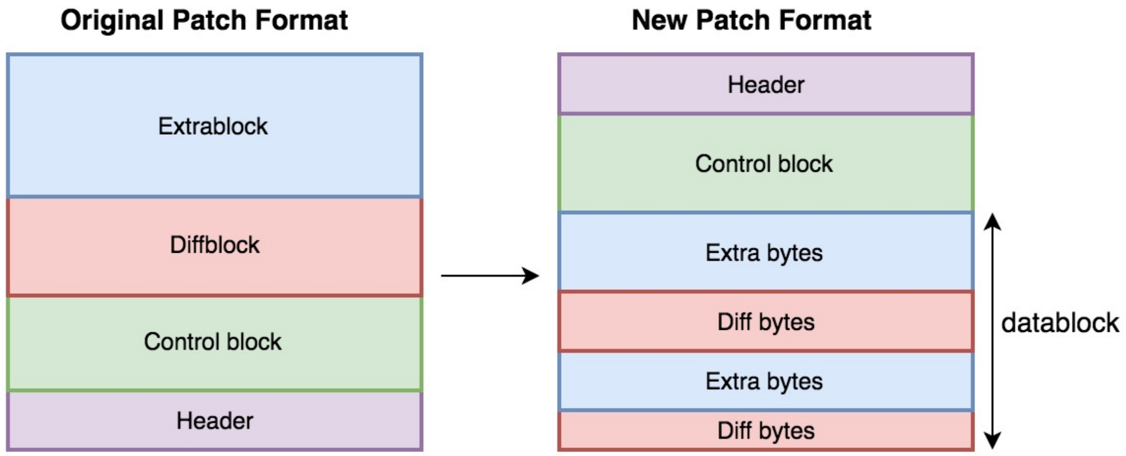

Figure 10: Patch structure comparison.

Seemingly, the incremental update process may now take place.

3.4 Process Overview

As the primary challenges to developing an incremental update mechanism within a resource-constrained environment have now been seemingly overcome, the update process may be illustrated from start to finish.

The first step to applying the received patch is to build a page of new firmware in RAM according to BSDiff's patching process. This is illustrated in Figure 11.

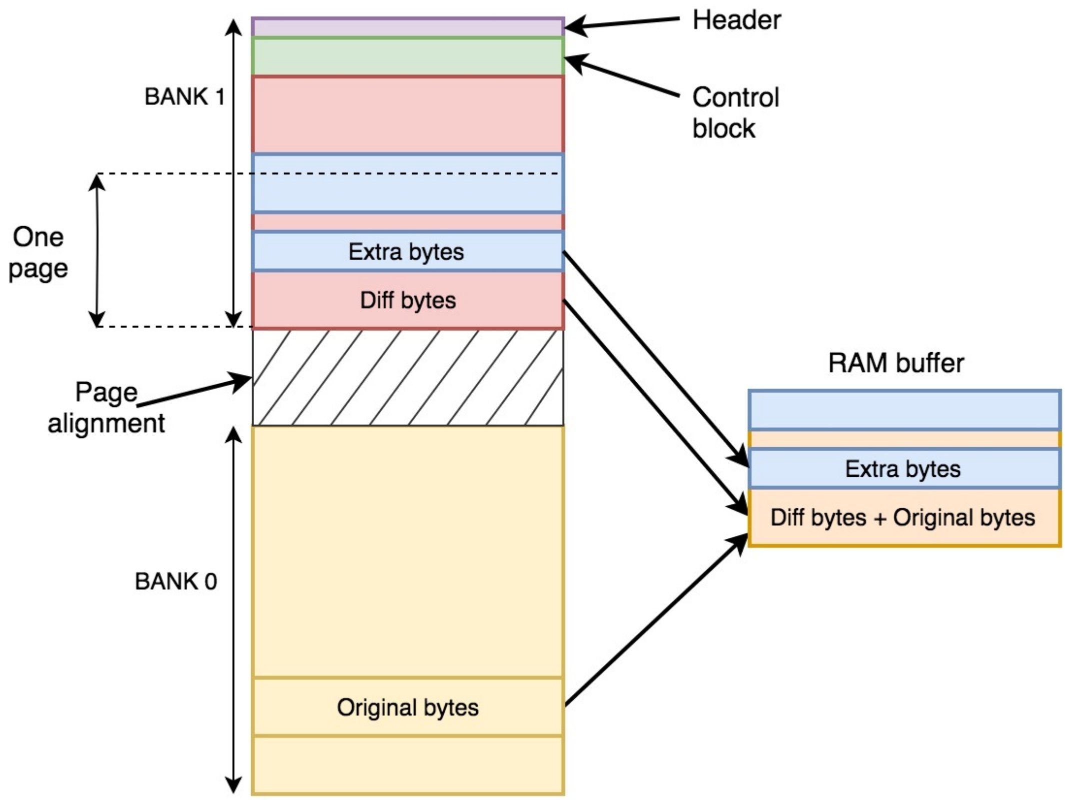

Figure 11: Forming a page of patch. The orange color within the RAM buffer illustrates the resulting new firmware is formed from adding the yellow old firmware bytes and red bytewise difference bytes together.

As the maximum amount of RAM may be limited to one page, on the basis that it makes sense to have a buffer size equivalent to the flash erasure size, part of one “ADD” or “INSERT” instruction may have to be split into multiple parts, on account of accumulated instruction bytes not aligning to a page boundary. This is shown in Figure 11 where the extra bytes segment is cut in two by the dashed line.

In any case, once the new page has been formed in memory, it’s time to save the first page of old firmware bytes into the first page of the patch data block which has already been used to form the new page in RAM. Once this is done, the new firmware page may then be written back into the first page of the old firmware. This is illustrated in Figure 12.

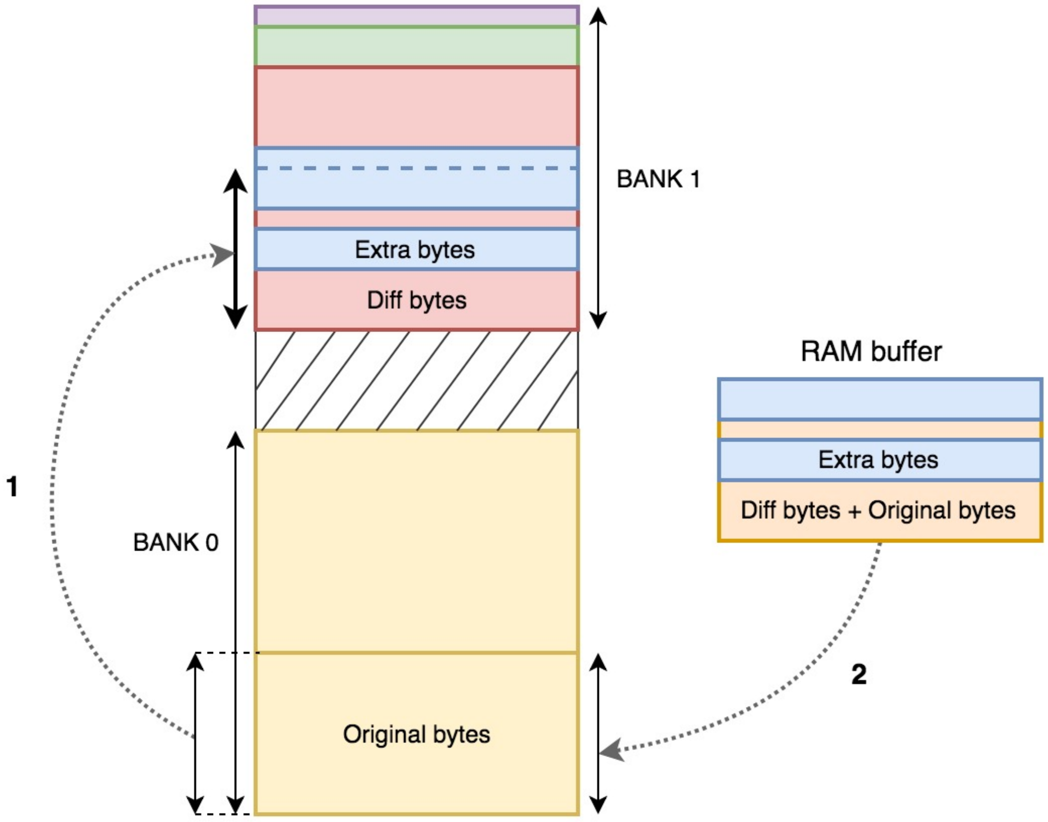

Figure 12: Saving a page of old firmware. The numbers '1' and '2' refer to the order in which the operations are carried out.

Figure 13 represents the state of memory having written the first page of new firmware.

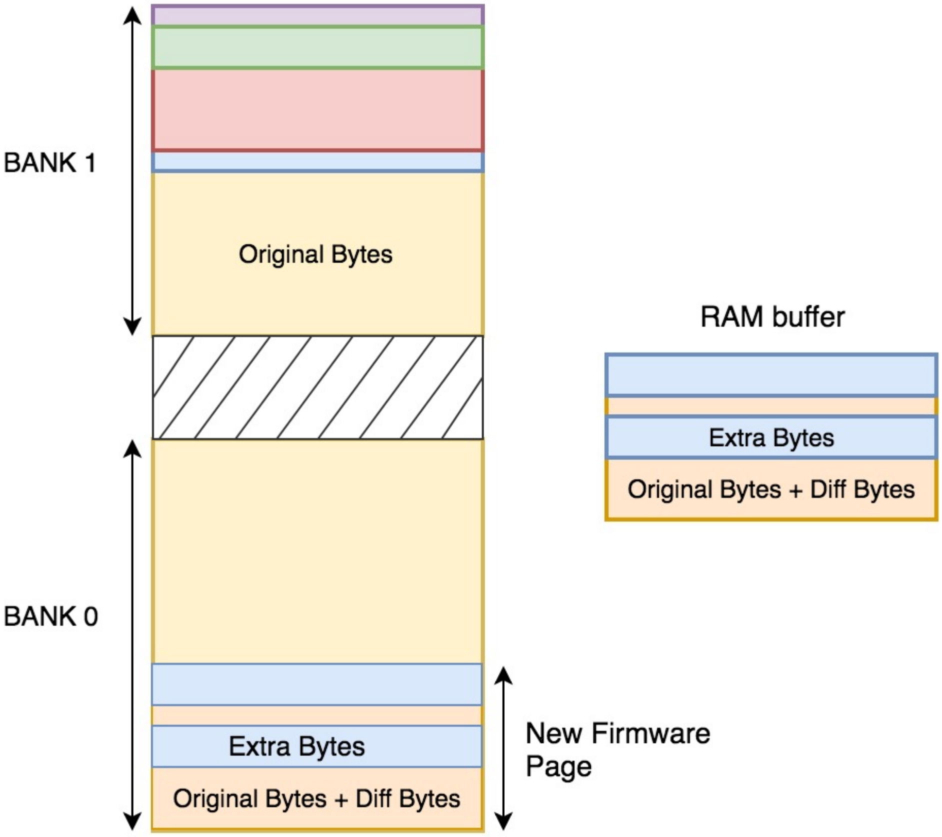

Figure 13: Final flash state after storing patched page.

To complete the whole patching process, this cycle of forming pages of patch in RAM, saving old firmware, and writing the new page back into the old firmware may need to take place repeatedly. Specifically, the amount of times it will take place is:

$$sizeOf(new\,firmware)/sizeOf(page)\ +\ sizeOf(new\,firmware)\%sizeOf(page)$$

Furthermore, when patching together new pages of firmware after the first has been written to flash, the patching process may then refer to old firmware bytes within either the patch or the old firmware bank, depending on whether they have relocated to the patch.

In the last cycle, the size of the new firmware to be written will likely not be a whole page, since it’s unlikely that the size of the new firmware is a multiplicative factor of a page size. Therefore, part of memory not representing the new firmware will simply be left erased; functioning as page alignment in essence.

This concludes the design of the incremental update mechanism proposed in the thesis, and makes way for discussion on a prototype implementation.

4 Implementation

"Action is eloquence."

– William Shakespeare

This chapter details the implementation of the design proposed earlier in this thesis, and therein discusses the main body of work carried out in the project, and the development challenges faced in creating the incremental design prototype. Much more detail of the actual update process not included in the Design, or previous chapters, is included in the Implementation.

Furthermore, each section relates to a significant and specific part of the implementation. For each section, a list of development steps are given, as well as a short discussion on the challenges faced, located under an unnumbered heading "*Challenges"*.

Section 4.1 first details the necessary steps to carrying out monolithic updates using *bootloader* firmware, supplied by the NRF5 SDK, and *nrfutil*: a command line tool for both creating monolithic update packages and transmitting them wirelessly over BLE.

Following on from this, the alterations to nrfutil to allow for creating and sending patch update packages are given in section 4.2, which include recompilation of serialization libraries to support updated metadata, known as an *init packet,* sent prior to firmware content.

The extensions to the bootloader firmware to receive incremental updates are discussed in section 4.3, and the steps required to reimplement BSDiff according to the design proposed are given in section 4.4.

Finally, the appropriate measures taken to port altered BSDiff to the bootloader are listed comprehensively in section 4.5. This section also contains a discussion on how overlapping of the new patched firmware and the patch itself is avoided despite the two being located directly above and below each other respectively.

4.1 Regular DFU

As the strategy taken to implement incremental methods consisted of working from firmware that supports monolithic updates, the first step was to carry out a regular monolithic update. Having configured a local copy of NRF5, and built nrfutil from source, the main steps required are to:

1. Flash the bootloader onto the device using *nrfjprog;* a command line tool for interfacing with Nordic Semiconductor development kits.
2. Flash a software defined BLE stack, also supplied by NRF5, in a similar manner as step 1.

3. Generate update packages using nrfutil. This requires supplying the tool with a firmware image, and private key used to verify the update within the bootloader.
4. Transmit an update. Firstly, this requires the board enter bootloader mode, thus exiting out of currently running firmware, and is done through holding down two on-board buttons while resetting the device. Secondly, another board with BLE support is then used to transmit the monolithic package.

Challenges

The main challenge faced in this section was dealing with errors raised from a BLE python package in nrfutil, and were mysteriously solved through cleaning up versions of Python and using a virtual environment.

4.2 Nrfutil Alterations

The following changes to nrfutil were necessary to support both generating and transmitting incremental patch packages:

1. Add a command line options for specifying a patch to be used instead of a firmware image.
2. The init packet must allow for including patch related metadata. These include specifying that the update is incremental and not monolithic, and the patch size included in the update.

3. Internally, code which handles conversion from a “.hex” executable format to “.bin” format, as well as padding these binaries to 4 byte divisible sizes, must be bypassed. The manifest file, which specifies the contents of an update package, may then be updated to include the patch file instead of firmware image.

Furthermore, generated code for the serialization of the init packet, created using *Protobuf*, must be recompiled using a command line tool, *protoc*, which may be provided by *brew*.

Challenges

In terms of development challenges, one was adding patch cases throughout the medium-sized nrfutil source, as the code is written in a relatively non-generic manner which makes reference to “firmware” or “binary” within variable names as much as possible.

Furthermore, if a board didn’t have the correct softdevice flashed onto it, an error related to BLE packet transmission would be raised. These sorts of behaviour tended to be figured out through some amount of reasoning and debugging via PDB; which was very useful.

4.3 Receive Incremental Updates

The changes required to receive an incremental update, without yet being able to patch it, include the following:

- Adding a case for incremental updates within init packet prevalidation checks, before the main firmware content is sent, to make sure that the init packet contains a patch size value.
- Compiling Protobuf serialisation based on nrfutil's new serialization: a tool is supplied by NRF5 for generating the necessary C and header files.

Challenges

Stepping through code on the bootloader is impossible due to BLE timeouts which interrupt and exit the board from bootloader mode. Despite this, debug messages may be used.

Furthermore, in the case of init packet serialization issues, the external Protobuf generated code had to be debugged. Before realizing that error messages set in Protobuf had been discluded in NRF5 SDK config to optimize compilation, this debugging was arduous.

4.4 BSDiff Alterations

The next significant part to the implementation consists of altering BSDiff according to the Design chapter. The changes to the patch generation are:

- Merging the diff block and extra block into one data block, such that they are ordered according to their position in new firmware.
- Relocating the header and control block to the top of the patch.

The changes to the patch application are:

1. Alter the program pointers which locate the header and control block, and use one pointer into the data block instead of two separate pointers for the diff block and extra block.
2. Apply patches on a page-by-page basis. This requires allocating a page sized buffer of memory, and breaking up command instructions into multiple instructions if the whole instruction leads to buffer overflow. For example, there may be only 10 bytes left in the current page buffer, but the next “INSERT” instruction is to insert 12 bytes. Therefore, the instruction is broken up so that the first 10 bytes are inserted into the buffer, the contents of the buffer are written, and the last 2 bytes are inserted at the start of the page buffer.
3. Save old firmware into the patch and write the new patched firmware back into the same position in memory as the old firmware; as per the design. Must make

sure that old firmware is read from the correct location; depending on whether it's stored in the overwritten patch, is in its original location, or partly both.

Furthermore, no padding of patches to fit a 4-byte / 32-bit size is required, although flash writes must be 4-byte aligned. The reasons for this are twofold:

- NRF5 SDK compiled firmware is padded to a 4-byte divisible size.
- The data block is the same size as the firmware; control statements are 24 bytes long, and the header is 32: all of which are 4-byte divisible.

Challenges

The first challenge in implementing the altered BSDiff was design related, and involved misinterpreting how the algorithm worked. The misconception was that areas of old firmware used in “ADD” instructions would be referred to sequentially, such that, for example, after the second page of old firmware had been used to form the new firmware, the first page of old firmware would no longer be needed.

Secondly, whenever there were differences between the patched new firmware and the actual new firmware used to create the patch, the reasons for the differences could be hard to determine. Use of LLDB to debug BSDiff was helpful, especially when setting various conditional breakpoints, but sometimes comparing the contents of the patched firmware to the original new firmware was necessary. In making these changes again, it would have been best to add a small testing framework.

4.5 Bootloader BSDiff Port

The final part of the implementation is to port the altered BSDiff patching to the bootloader, which constituted the following:

- Add altered BSDiff as an external library.
- Call the patching function with the necessary pointers into the start of the old firmware and the start of the patch.
- Before writing new patched firmware, or saving old bytes, erase the necessary pages.
- Enable NRF5's flash module, which provides a version of malloc, to allow the allocation of a page of RAM. Furthermore, the size of the maximum allocatable contiguous block of memory had to be enabled within the SDK's memory manager module.
- Change numerous types used locally within BSDiff that aren't natively supported by NRF5, like "off_t" types to "int64_t" types. Furthermore, some thinking around implicit type casts was necessary, due to, for example, BSDiff using single byte "uchar" pointer addressing and NRF5 using "uint32_t" addressing.

Challenges

The challenges in porting BSDiff mainly consisted of determining the reasons for flash errors and memory allocation errors, increasing the default module log levels to determine what error codes were returned, whose definitions could be *grepped* within the SDK.

Furthermore, determining the addresses of banks of memory which store the patch and the current firmware was more difficult than suspected as they were dynamically allocated in software depending on the size of firmware.

Patch-Firmware Collision

Although the Design specifies the patch be located above the old firmware in memory, the location of these two banks of memory is arbitrary. However, if they are located below and above each other respectfully, then to make sure the patch and the new patched firmware don't overlap with each other, two cases need to be considered.

The first of these is when new firmware is smaller than the current firmware on the device. Therefore, it won't overlap with old firmware if the patch is located directly above the old binary. This is illustrated in Figure 14.

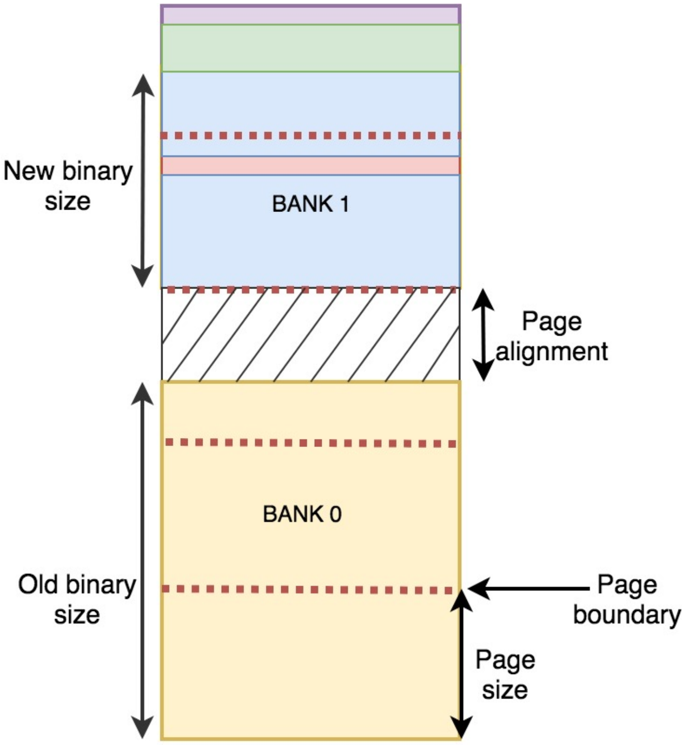

Figure 14. Patch located just above old firmware. The patch is situated on the next page boundary after the old firmware.

In the second case, the new firmware is greater in size than the old firmware, meaning that if the patch is situated directly above the old firmware, then it may be partly overwritten by the new firmware. Therefore, the patch should be located at an offset greater than the size of the new firmware from the start of the old firmware, illustrated in Figure 15.

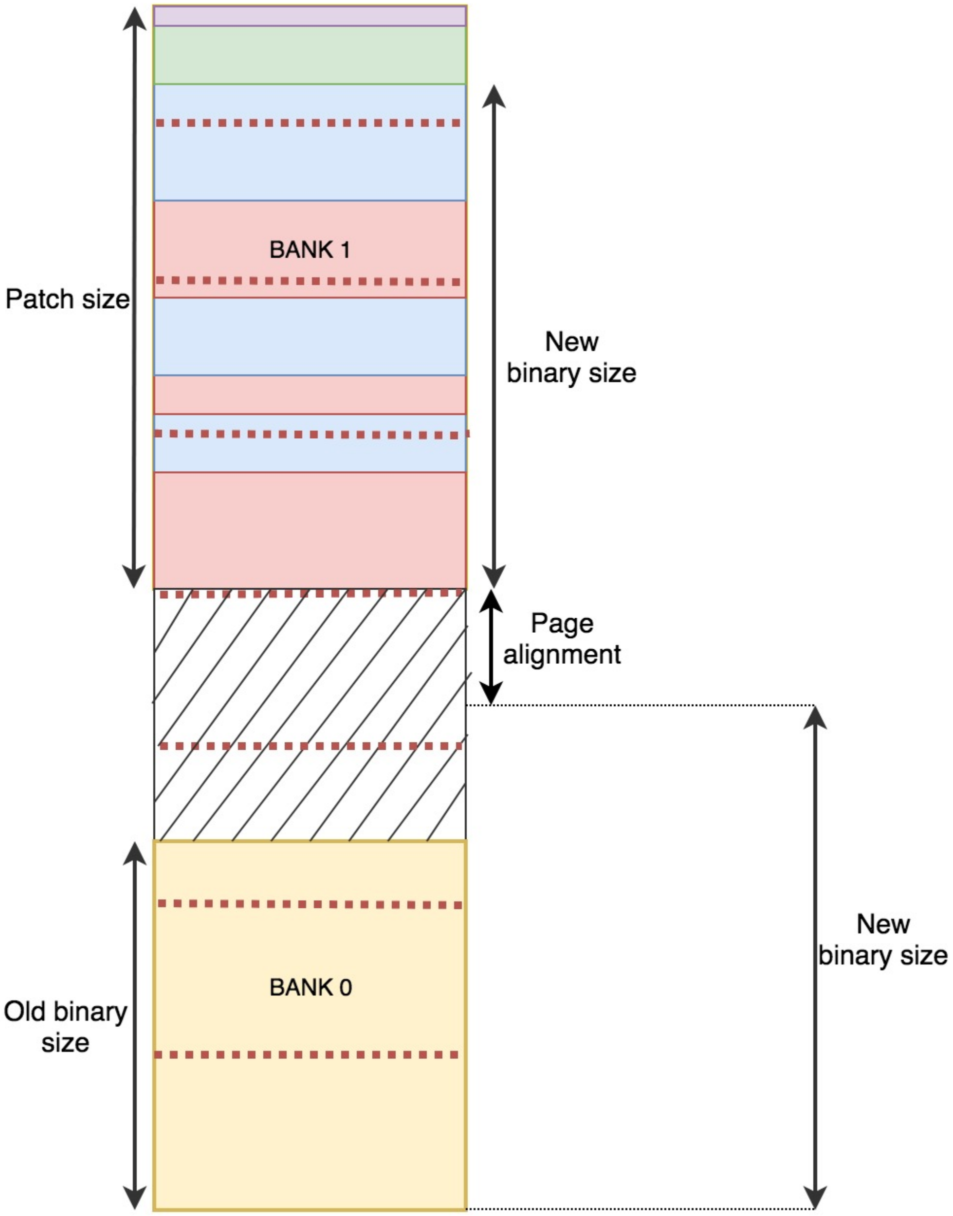

Figure 15: Patch located above new firmware.

5 Results

"It is the mark of a truly intelligent person to be moved by statistics."

– George Bernard Shaw

5.1 Experimental Setup

The experiments carried out consist of transmitting and applying incremental updates according to the manner outlined in the Design and Implementation chapters to a Nordic Semiconductor nrf52832 development kit. The board is flashed with both the altered bootloader firmware outlined in the Implementation chapter, and a software defined BLE stack known as the *s130* softdevice in the NRF5 SDK.

The test cases used are based on "Blinky" application firmware supplied in NRF5 SDK version 14.2.0.

The Blinky application uses an infinite *while* loop with an inner *for* loop to loop through each LED on the board and turn it on; wait for an arbitrary period; and turn it off. The manner in which the firmware is altered in each version corresponds to the following:

1. v1 is the same as the original SDK supplied version.
2. v2 removes the outer infinite *while* loop and instead replicates the inner *for* loop once, such that there are two *for* loops.

3. v3 is the same as the original but uses a variable to set the initial waiting duration and increments same after each iteration of the outer *while* loop.

The size of each version is given in Table 1.

Blinky version	Size (bytes)
v1	2476
v2	2920
v3	3264

Table 1: Blinky version sizes.

Patches are then generated to go from version ***n*** to ***n+1***; shown in Table 2. They and are compressed with FastLZ, as suggested by [3], which was chosen from a selection of other compression libraries tested on BSDiff which included huffman and lz77 compression.

Patch	Size (bytes)
v2	237
v3	427

Table 2: Compressed patch sizes.

5.2 Data

In comparison to monolithic update mechanisms which use compression, the only supplementary stage to incremental updates is the intermediate patching of new firmware pages. The extra CPU time incurred by this patching - directly corresponding to additional energy consumption - for the three Blinky test cases can be seen in Table 3, which also includes time taken to apply the corresponding monolithic updates.

	Monolithic	Patch
v2	70.4	122.0
v3	76.8	135.4

Table 3: Update application times (ms).

Despite apparent large percentage differences between the time taken to apply an update incrementally and monolithically, it's important to see the actual average difference is 51ms. In comparison, due to high compressibility of patches displayed in Table 2, patching saves 640ms off the average of transmission times relative to compressed whole images - shown in Table 4.

	Monolithic	Patch
v2	1260.1	618.4
v3	1275	636.7

Table 4: Transmission times (ms)

Therefore, given that RF transceiver active-time typically requires 10 times the amount of energy than CPU active-time [6], the energy expense due to patching is negligible.

Quantifiably, various test cases - spanning from the addition of new applications to application upgrades - used to evaluate FastLZ compressed BSDiff in [3] show energy saving which are at least 30% using monolithic updates. Although patch reformatting used in this thesis to accommodate highly memory-constrained devices may affect update sizes, it's clear that an average of less than 0.5% for the test cases, shown in Table 5, is also negligible, and thus: practically identical energy savings would be expected.

Patch	Size Increase
v2	0.84%
v3	0%

Table 5: Compressed reformatted patch size percentage differences.

Discussion

In terms of explaining these small increases in patch sizes: they are due to merging of the diff block and extra block into one data block. The primary way that BSDiff allows for such high compression is long consecutive sequences of zeros that correspond to exact matching regions between two versions. Although no single exact match is split through the redesign, two matches which were located contiguously may be separated and resituated in different parts of the data block. Theoretically, this could result in even longer sequence of zeros. However, in practice, it seems it may increase the compressed size; by a negligible amount.

7 Conclusions & Future work

“The possession of knowledge does not kill the sense of wonder and mystery. There is always more mystery..”

– Anais Nin

7.1 Conclusions

One primary conclusion of this thesis is incrementally updating highly memory-and-resource-constrained devices is possible through the update mechanism outlined in the Design chapter. Furthermore, due to energy saved in transmitting smaller updates, and the negligible patch size increase due to reformatting, the incremental mechanism should tend to result in significant energy savings.

Interestingly, the incremental mechanism requires no extra flash other than that required to receive the patch, and a small amount of RAM to buffer a page of patched firmware. This ensures that the maximum update size remains as large as possible.

7.2 Future Work

7.2.1 Decompression

BSDiff requires patches be compressed to result in file size savings. Without compression, patches are actually larger than the new firmware to be patched together, since the patch file is the same size as the original firmware image plus the size of the control statements and header. Despite this, decompression is still only an intermediate step in the update process, and would be relatively easy to implement from a conceptual point of view.

One means of carrying out decompression would be to receive the compressed patch at the top of a bank the same size as the decompressed patch, produce decompressed patch on a page-by-page basis in RAM, and write the patch pages in an ascending manner starting from the bottom of the bank. This is illustrated in Figure 16.

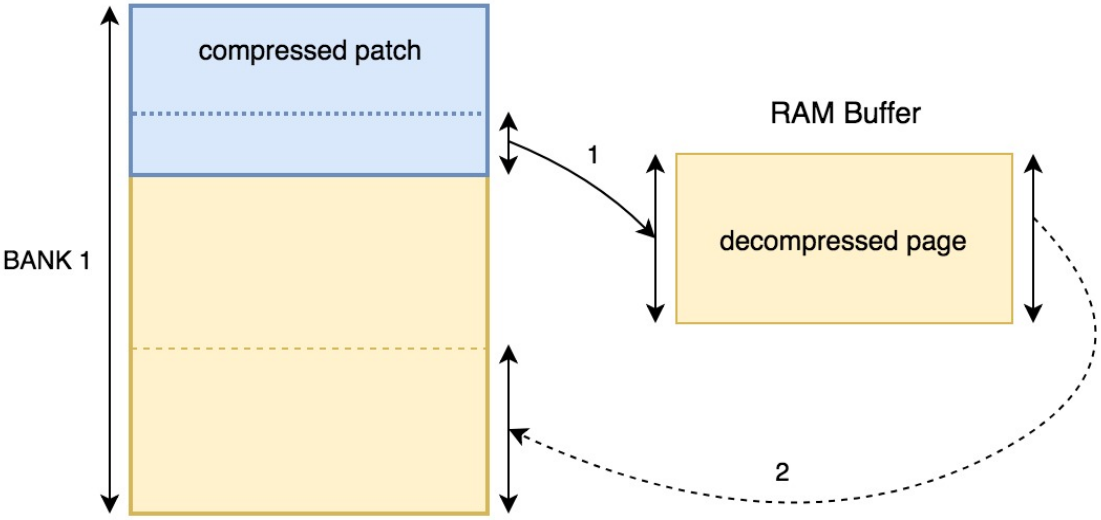

Figure 16: Decompression. '1' represents the inflation of compressed patch to produce one page of decompressed patch. '2' represents the writing of a decompressed page to flash.

This process continues until the whole patch has been decompressed and written back into persistent storage. This means the last page of decompressed patch, or the final pages of decompressed patch, will overwrite the compressed patch.

In order for this to work, the size of the decompressed patch would need to be known beforehand such that the size of the bank may be set accordingly. One way of achieving this would be including the decompressed size within update metadata.

To assess which library would be best to use, considerations of memory footprint, processing requirements and compression ratios of numerous libraries would need to tested. However, this has seemingly been carried out to a great extent in [3, 4], concluding that FastLZ or lz77 is best to use along with BSDiff.

7.2.2 Robustness

There is a greater amount of security in update processes that are robust and reliable. Robustness hasn’t been focused on to a great extent in this thesis, and therefore, there are two ways in which it could be increased significantly and relatively easily.

The first of these means would be saving a small amount of state regarding the patching process, such that, if a device was to run out of energy or be unplugged, that it could resume patching once it was powered back on. However, one issue with the current design is if a device ran out of energy after erasing a page of used patch in preparation to save an old firmware page, then the patched firmware page in RAM about to overwrite the original old firmware page would not be reclaimable.

Therefore, one solution is to preserve a page of flash storage - available to the update process - to intermediately store the patched firmware page in RAM before erasing the used patch page. Despite this, the maximum update size is consequently reduced. In any case, the state needed for this process would correspond to a pointer into the last page of patched firmware successfully written into its final location over the old firmware.

The second means of increasing robustness could be verifying that updates are meant to be applied to the version of firmware on a board. This can be done by including what version of firmware is expected on the board in update metadata. The bootloader may then compare the firmware version on the board with this incoming version before receiving the update.

7.2.3 Efficiency

There are many ways in which the design could be made more efficient in terms of reducing energy consumption and reducing the number of flash write cycles; as flash memory may only support a limited number of write cycles at any given address.

Firstly, when patching new firmware, if a page matches the corresponding page of current firmware, then the new page shouldn't be written. This could be detected relatively easily by determining that there are no new bytes in a patched page and that the sum of diff bytes for the page is also zero. Furthermore, this detection could be carried out whilst patching firmware or in the patch generation if the functionality of control statements is extended to include skipping page writes.

In terms of flash longevity, one simple means of increasing such would be to make use of a circular writing system; much like a circular buffer. This would ensure that all the flash available is written to approximately an equal amount of times, thus increasing the lifetime of the device. However, such a circular system would also necessitate keeping state for where firmware is stored, and hence is slightly complex.

Lastly, reversing updates, when required, could be done more efficiently than sending a reverse patch through using the old firmware saved during patching processes. Furthermore, as some of the original firmware may be not have been overwritten by the new patched firmware, and is situated just before where the patch was received, the process would simply constitute copying over the old firmware, saved in the patch file, back to its original position.

Citations

1. “Gemalto and Cargo Tracck team up for a Successful Sting Operation“, https://www.gemalto.com/m2m/customer-cases/iot-protects-rainforest, [Accessed 8 May 2018]
2. S. Brown, C. J. S. (2006). "Updating Software in Wireless Sensor Networks: A Survey."
3. Stolikj, M. C., P.J.L.; Lukkien, J.J. (2012). "Efficient reprogramming of sensor networks using incremental updates and data compression."
4. Mei-Ling Chiang, T.-L. L. (2011). "Two-Stage Diff: An Efficient Dynamic SoftwareUpdate Mechanism for Wireless Sensor Networks ".
5. S. Brown*, C. J. S. "AN ENERGY BENCHMARK FOR SOFTWARE UPDATES ON WIRELESS SENSOR NODES."
6. Milosh Stolikj, P. J. L. C., and Johan J. Lukkien "Efficient reprogramming of wireless sensor networks using incremental updates."
7. La Manna, V. P. (2012). "Local dynamic update for component-based distributed systems."
8. Chen Rong, C. H.-b., Zhang Feng-zhe, Zang Bin-yu (2007). "Dynamic update of operating systems."
9. Xiaohui Xu, L. H., Dejun Wang (2007). "Supporting dynamic updates of componentized service."
10. Gregersen, A. R. (2011). "Implications of modular systems on dynamic updating."

11. Waqaas Munawar, M. H. A., Olaf Landsiedel, Klaus Wehrle (2010). "Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks."
12. Culler, P. L. a. D. "Maté: A Tiny Virtual Machine for Sensor Networks ".
13. George Oikonomou, I. P. (2011). "Experiences from Porting the Contiki Operating System to a Popular Hardware Platform."
14. Niels Reijers, K. L. (2003). "Efficient Code Distribution in Wireless Sensor Networks."
15. Jaein JEONG , D. C. (2009). "Incremental Network Programming for Wireless Sensors."
16. Rajesh Krishna Panta, Saurabh Bagchi and Samuel P. Midkiff (2009). "Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation."
17. "BSDiff", http://www.daemonology.net/papers/bsdiff.pdf, [Accessed 8 May 2018]
18. Alsanussi, M. A. M. a. R. A. (2017). "Review of Methods Used in Computer Science Research."
19. SADAKANE, N. J. L. A. K. (1999). "FASTER SUFFIX SORTING."