University of Reading

SlimFS: a thin and unobtrusive file system for embedded systems and consumer products

Article

Accepted Version

Boubriak, A., Cooper, A., Hossack, C., Permogorov, D. and Sherratt, R. S. ORCID: https://orcid.org/0000-0001-7899-4445 (2018) SlimFS: a thin and unobtrusive file system for embedded systems and consumer products. IEEE Transactions on Consumer Electronics, 64 (3). pp. 334-338. ISSN 0098-3063 doi: 10.1109/TCE.2018.2867826 Available at https://centaur.reading.ac.uk/78875/

It is advisable to refer to the publisher's version if you intend to cite from the work. See Guidance on citing.

To link to this article DOI: http://dx.doi.org/10.1109/TCE.2018.2867826

Publisher: IEEE

All outputs in CentAUR are protected by Intellectual Property Rights law, including copyright law. Copyright and IPR is retained by the creators or other copyright holders. Terms and conditions for use of this material are defined in the End User Agreement.

www.reading.ac.uk/centaur

University of Reading

CentAUR

Central Archive at the University of Reading

Reading's research outputs online

Full-Text version

Title: SlimFS: A Thin and Unobtrusive File System for Embedded Systems and Consumer Products

Authors:

Andriy Boubriak, RedWave Labs, Harwell, OX11 0QG UK, aboubriak@redwavelabs.com

Adam Cooper, was with RedWave Labs, Harwell, OX11 0QG UK. He is now with YOOX Net-a-Porter Group, LONDON, W12 7FP, UK, adam.cooper@ynap.com

Christopher Hossack, RedWave Labs, Harwell, OX11 0QG UK, chossack@redwavelabs.com

Dmitri Permogorov, RedWave Labs, Harwell, OX11 0QG UK, dpermogorov@redwavelabs.com

R. Simon Sherratt, *Fellow, IEEE*, Biomedical Engineering Department, University of Reading, RG6 6AY, UK, sherratt@ieee.org

Funding:
This work was supported in part by a Knowledge Transfer Partnership grant #10003 and RedWave Labs, Harwell, UK.

Abstract:

It is common place for embedded systems and consumer products to contain flash memory for nonvolatile storage. While there are many applications that require the data stored in the flash memory to be in a given structure enabling the data to be externally accessed, there are also many embedded consumer applications where the content of the flash memory is only accessed locally. In this case, the local application can benefit from having a minimized bespoke file system optimized for the application, resulting in lower power and faster access speed than using public file systems.

This paper analyses the overhead in using the commonly used File Allocation Table File System (FatFS), and proposes a significantly faster, smaller footprint, and hence lower power file system, termed SlimFS. The work has clear applications to low power embedded consumer applications, specifically battery driven wearable devices for healthcare and 'green' electronic systems.

Publication: IEEE Transactions on Consumer Electronics
Publisher: IEEE
ISSN: 0098-3063
DOI: 10.1109/TCE.2018.2867826
Volume: not yet assigned
Issue: not yet assigned
Accepted Date: 17th August 2018
pp.: not yet assigned

Key words: Flash memories, Embedded system, Wearable sensors, Low-power electronics, Public healthcare

I. INTRODUCTION

Many Consumer products are created from embedded processor devices that make use of flash based NAND memory [1]. Consumer applications may store data to their local flash using the File Allocation Table File System (FatFS) to maintain compatibility with typical Personal Computers (PCs), allowing consumers to utilize and share data across their devices. Therefore, to improve the performance of FAT file systems for embedded devices, previous work has focused on "flash-aware" techniques. These techniques take into consideration the technology differences of flash memory compared with traditional data storage mediums, typically Hard Disk Drives (HDDs). They are also often further optimized by making assumptions about typical workloads of such flash-based devices particularly for media players. However, there are many consumer devices with embedded flash memory (e.g. home controllers, simple IoT sensors, simple wearable health devices including pedometers) that have no need to share the data in their flash memory, and while it is tempting to use a common file system for ease of development, this paper will present that such indolent design methodologies can result in compromising performance, particularly for battery driven low-power systems, e.g. wearable devices for public and consumer healthcare.

File systems provide a way of easily storing and accessing multiple sets of data without the user having to keep track of memory locations. Data is typically stored in the device with the location of the data referenced in a storage table. File systems provide a specific structure that enables the user, or application, to reference the data in the memory using metadata in the storage table. In the present work, a 'file' is considered as data with metadata attached to it.

This paper analyses the most common file system, FatFS, identifies performance issues and presents an alternate file system, termed SlimFS, that creates a highly minimized file system specifically designed to present a low Central Processor Unit (CPU) load, resulting in lower power usage profile compared to FatFS, ideal for implementing in consumer based embedded systems and the Internet of Things (IoT).

The structure of the paper is as follows; Section II presents the literature survey, Section III presents our proposed minimized file system, Section IV presents the results and the conclusions are presented in Section V.

II. LITERATURE SEARCH

Flash devices contain a Flash Translation Layer (FLT), a software layer below the file system to abstract the characteristics of the implemented flash memory in order to emulate the behavior of a traditional HDD. In NAND based flash memory, the smallest granularity of read and write operations occur on a page level, typically in the range of 512 bytes to 8 kilobytes [1] [2], and when the flash memory stores data the page must be erased before being written to with the new data. The smallest erase operation however, erases at a block level, typically being 32 or 64 pages [2]. Erase operations are significantly slower than write operations. When writing to files the metadata in the FAT table is frequently updated. This requires an entire block to be erased and reprogrammed when only a 4-byte FAT entry (in the case of 32-bit FAT) is being modified. This operation is undesirable as flash memory can only undergo a finite number of Program/Erase (P/E) cycles before failure.

The improvements suggested by Kim and Shin [1], and Park and Ohm [3] are similar techniques which make assumptions concerning the workload. They are both flash aware techniques designed for a multimedia storage use case, such as mp3 players, photography or video recording. Therefore, the assumptions made are that the files are large, will only be written to once, and that only one file will be written to at any given time. The work Park and Ohm [3] takes advantage of this workload by reallocating all free clusters in the FAT table to the file when it is opened and then deallocates them when it is closed. This operation prevents the need for multiple program erase cycles to the FAT table as the file is being written to. Kim and Shin [1] used a similar technique, however they assumed a maximum file size and allocated enough clusters for that maximum file size.

Kim *et al.* [2] proposed a new file system, termed MNFS with multiple notable differences to the FAT file system, one of which is also motivated by the assumption that a typical use case is large multimedia files. This assumption allows them to reduce the number of metadata updates and P/E cycles by increasing the cluster allocation size to match the flash block size so that fewer entries in the allocation table need to be updated. This also makes the allocation table smaller.

Another common approach to reduce the program erase cycles in flash memory is to employ a log block-based FTL scheme such as FAST [4] and EAST [5]. In these schemes, when existing data within a file needs to be overwritten, rather than performing an erase on the entire block to rewrite a single page, new writes are written to log blocks. The previous data within a page is then invalidated by writing to the spare area of the block. The spare area of the block is an area NAND flash memory which usually contains metadata about the block or an Error Correcting Code (ECC). Periodically, log blocks are merged with existing data, merging multiple pages into a block at once, reducing the total number of required P/E cycles. This provided both an improved write performance and also increased the lifespan of the device.

Other research also exists on improving the FAT file system itself specifically for embedded systems rather than the underlying FTL. Munegowda *et al.* [6] proposed a *directory compaction* technique. The directory information metadata used to build the file system was stored in clusters which are allocated within the FAT table like normal data. When all directories and files that are

contained within a single metadata cluster are deleted, that cluster in the FAT table is freed. This technique makes the metadata required for the FAT file system smaller, leaving more space for working data.

Research on improving the FAT file system more generally is the caching system proposed by Hwang and Won [7] for storage devices with low data transfer rates. When performing multiple sequential writes to a file, a system utilizing the normal FAT architecture must occasionally perform a read request of the FAT metadata. The read is performed to find an unallocated cluster which can be allocated to the file, as the memory usage increased. The system proposed by Hwang and Won caches the entire FAT table. This prevents the need for additional read requests during a sequential write operation and, therefore, decreases latency.

As files are created and deleted, the free clusters can also become spread out across the FAT table, creating irregular response times when reading the FAT table to find a free cluster. Choi *et al.* [8] sought to specifically address the issue of irregular response times, without having to cache the entire FAT table. They considered a *cluster bank* structure instead of a FAT system. A cluster bank consists of many cluster stack structures which themselves contain the cluster allocation data. A cluster stack consists of a cluster group and a cluster table. The cluster table is an 8x8 table which represents clusters and whether or not they have been allocated. The cluster group is a byte where each bit represents which rows within the cluster table are fully allocated. The software stores, within a register, the first cluster stack containing a free cluster. Using this information, it then reads the cluster group within the specified cluster bank to find the lowest row within the cluster table which contains a free cluster. Then it reads that row in the cluster table to find the lowest available cluster and it then updates all the cluster bank information at the end of the allocation. The cluster stacks only inform the system whether a cluster is free or allocated, with no information about the next cluster belonging to the file as with FAT. There is a directory section, as with FAT, which stores the first clusters allocated for each file. The next cluster allocated to the file is then stored with the data for the file itself. The cluster bank method eventually requires marginally more storage than the FAT file system, however it also provides a fixed number of reads for free cluster allocation. This property is useful for writing large multimedia files.

The SlimFS system proposed in this paper is inspired by the literature, but is designed considering the needs of typical embedded applications, e.g. small IoT based devices, sensor systems, embedded controllers, etc. However, unlike much of the previous work typically concerned with multimedia storage, we focus on tiny embedded systems which read and write to the flash memory in an *a priori* structure.

III. PROPOSED FS

This section presents a review of the implementation of FatFS, and then presents our SlimFS solution.

A. Existing FatFS

The main feature that defines FAT file systems is the File Allocation Table (FAT). Disk drives are split into blocks/clusters, and the FAT table is used to determine (point to) which blocks join together to form the requested data or file. The file system has a root directory, which is stored at a predetermined location, typically the first block. As can be seen from Fig. 1, a file or directory can be found by recovering data from the root directory. Blocks containing directory information store metadata about each file/directory and the location of their starting block. The FAT table is then used to find the location of remaining blocks where the file/directory continues. As a way of illustrating by example, Fig. 2 presents a typical FatFS structure to illustrate the start and end blocks for files. The root directory indicates the starting block for 'File 1' is block 3. Block 3 in the FAT table indicates that the file continues in block 4 (i.e. 0004). The FAT table then indicates that the file ends in block 4, (i.e. FFFF).

File data is not always stored in contiguous clusters. For example, if two files are created and stored in adjacent clusters, but at a later stage the first file gets data appended, then the next cluster is already allocated and the file system will have to find another free cluster to store the rest of the data. The clusters are chosen by the file system using various algorithms and allocation methods.

FatFS is a file system layer that is independent of the application or storage device, as can be seen in Fig. 3. It provides an Application Programming Interface (API) for the application layer with functionality including such features as opening and creating files and directories, reading and writing, closing, navigating…etc. A set of media access functions need to be provided to interface with the actual storage device. These have to be programmed to interface to lower level drivers specific to the storage device i.e. Secure Digital (SD) card or flash driver.

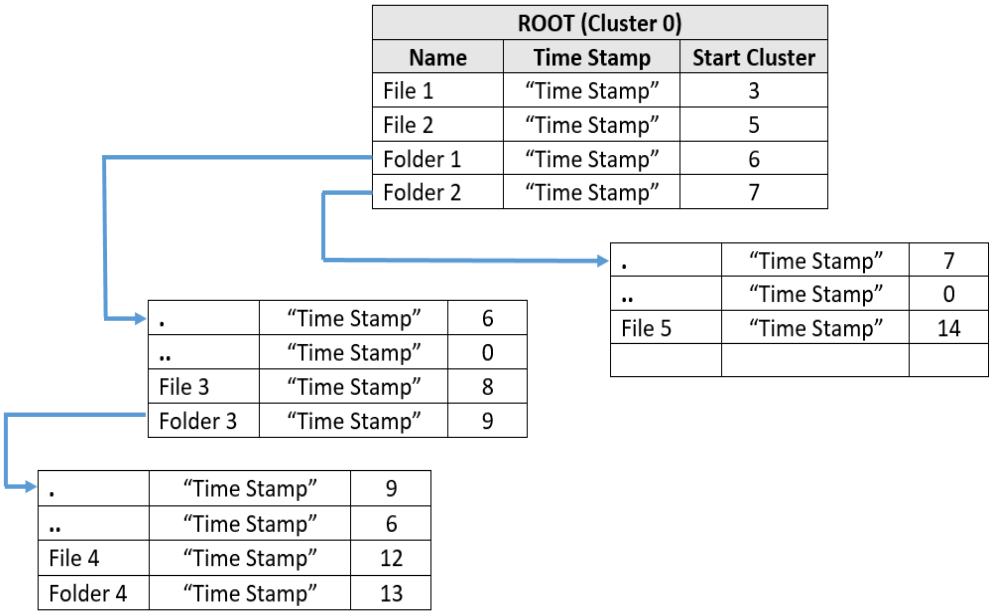

Fig. 1. – FatFS file look-up mechanism.

0 (ROOT)	1	2	3
FFFF	0000	0000	0004
4	**5**	**6**	**7**
FFFF	FFFF	FFFF	FFFF
8	**9**	**10**	**11**
000A	FFFF	000B	FFFF
12	**13**	**14**	**15**
FFFF	FFFF	FFFF	0000

Fig. 2. – FatFS File Allocation Table (FAT).

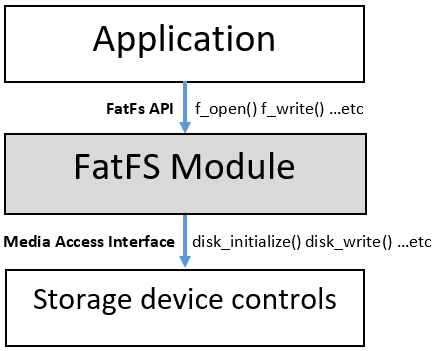

Fig. 3. – FatFS software layers.

B. Proposed SlimFS

SlimFs was created as an alternative way of storing data for embedded applications where the number of files required and their functionality is fixed. The requirements of the file system were:

1. Low memory consumption,
2. Only Open, Read, Write and Close operations,
3. Save binary data (saved data structures) and text,
4. Option for circular text files i.e. when the file is full, start overwriting from the start,
5. Corruption detection of critical data,
6. Interface with both EEPROM and flash devices.

The primary difference between SlimFs and FatFS is the memory allocation of the files. FatFS allows new files to be created at any time after initialization, however, for the purpose of embedded systems, it is not usually necessary to create files in such an unstructured order. An embedded device's role is very specific and any such files required by the device are known *a priori*. Furthermore, the embedded device's memory needs not to be accessed externally. Therefore in SlimFs, all the required files and their locations in memory are predetermined within the code. Initialization of the file system consists of creating pointers to these predetermined files. Thus removing the significant overhead process of the file system allocating memory, saving time and minimizing the amount of unused memory space.

As shown in Fig. 4, the structure of the new file system consists of 3 main parts, i) a global static file system class, ii) a file base class, and iii) subclasses of the base class for each of the 3 types of files required; binary data (saved data structures, typically for sensor reading), a circular text file, and a non-circular text file.

C. Implementation of SlimFS

All the required files are known *a priori* and their physical addresses may be hardcoded or enumerated. The file metadata is stored in a structure with the attributes of *File System Pointer, File Index, File Name*, *Physical Start Address* and *Physical End Address*. The type of file required is then created by instantiating its corresponding class of *binary*, *text* or *circular text*. Then, calling the *open(FILE INDEX)* function on the instance and passing the *FILE INDEX* to be associated with the new instance links them together. This adds the file to the file systems list of file pointers with all the relevant information.

Fig. 5 presents the SlimFS look up table. It can be seen that compared to FatFS, SlimFS has a significantly simpler table structure allowing for reduced memory and a significant speed increase.

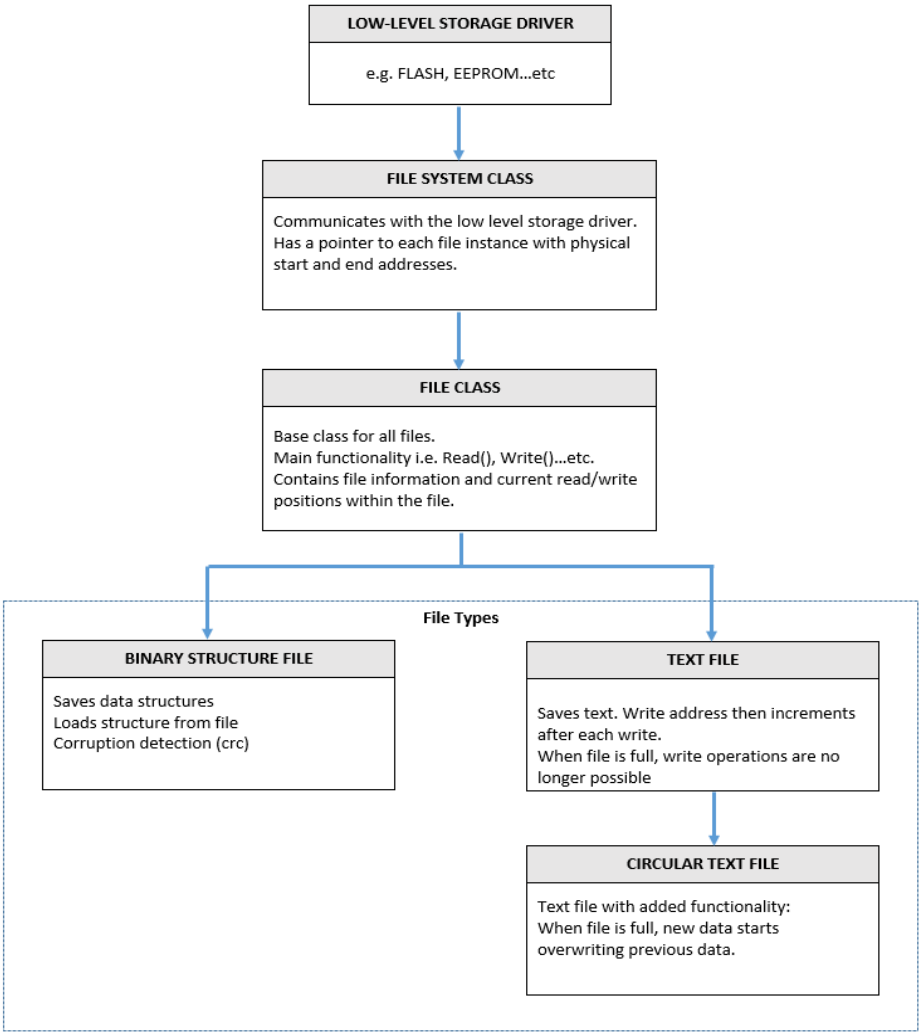

Fig. 4. – SlimFS software layers.

IV. RESULTS

Two test projects were made to compare the performance of both file systems using a common 32-bit 80MHz CPU. Each project was bare, only containing the required files to setup the file systems under a Real Time Operating System (RTOS) and interface with an off-the-shelf 8-Megabit flash memory IC [9].

Each project performed the following operations - initialization, creating/opening a file, writing 4096 bytes, reading 4096 bytes and closing the file. Physical timings were captured by toggling one of the microprocessor output pins either side of each operation and capturing with a counter/timer. The results are presented in TABLE I. As can be seen, SlimFS is significantly faster than FatFS in all areas, and therefore offers significant advantages over FatFS in reducing system power and storage delays in applications where the required files are known *a priori.*

The project codebase was stored in the microprocessor local flash memory. While the program memory in both projects was heavily dominated by the RTOS, and as can be seen from TABLE I, SlimFS required significantly smaller program memory than FatFS with a reduction of approximately 9kB. FatFS is much larger than SlimFS in part due to the optional features available that go unused.

The required runtime memory was stored in SRAM and FatFS used more RAM than SlimFS in part due to its use of caches. By default each FILE variable is 4096 bytes in size due to the inclusion of a cache, although this can be removed for further optimization.

“File 1”	“File 2”	“File 3”	“File 4”	“File 5”	…

File System	Index	Name	Start Address	End Address
&fs	FILE_1	“File 1”	0x0000	0x0FFF
&fs	FILE_2	“File 2”	0x1000	0x1FFF
&fs	FILE_3	“File 3”	0x2000	0x2FFF
&fs	FILE_4	“File 4”	0x3000	0x3FFF
&fs	FILE_5	“File 5”	0x4000	0x4FFF

Fig. 5. – SlimFS simplified look-up table.

TABLE I
COMPARISON RESULTS OF FATFS VS. SLIMFS

Feature	FatFS	SlimFS
File system Initialize	2.7 s	< 1 ms
Open new file	108ms	76 ms
Write 4096 bytes	176ms	107ms
Read 4096 bytes	140ms	33ms
Close file	312ms	N/A
Open new file, write and read 4096 bytes, then close	736ms	216 ms
Required program storage	55.298kB	46.172kB
Required runtime SRAM	60.600kB	43.364kB

V. CONCLUSIONS

This paper has presented an extremely lightweight file system, termed SlimFS, for use in consumer embedded systems and IoT. SlimFS has a significantly simplified look-up table that exploits typical system constraints found in embedded consumer systems. While SlimFS can be used in small embedded applications where the files that need to be opened and written to are known in advance, the profiling results presented show SlimFS to be significantly faster and consume less CPU resources than the popular FatFS.

By implementing SlimFS, consumer will see clear benefits because that devices will respond faster when data needs to be accessed in flash memory and battery lifetime can be extended as SlimFS reduces the CPU load compared to FatFS.

REFERENCES

[1] Y. Kim and D. Shin, “Improving file system performance and reliability of car digital video recorders,” *IEEE Trans. Consum. Electron.*, vol. CE-61, no. 2, pp. 222–229, May 2015, 10.1109/TCE.2015.7150597.

[2] H. Kim, Y. Won and S. Kang, “Embedded NAND flash file system for mobile multimedia devices,” *IEEE Trans. Consum. Electron.*, vol. CE-55, no. 2, pp. 545–552, May 2009, 10.1109/TCE.2009.5174420.

[3] S. Park and S.-Y. Ohm, “New techniques for real-time FAT file system in mobile multimedia devices,” *IEEE Trans. Consum. Electron.*, vol. CE-52, no. 1, pp. 1–9, Feb. 2006, 10.1109/TCE.2006.1605017.

[4] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park and H.-J. Song, “A log buffer-based flash translation layer using fully-associative sector translation,” *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, Article 18, July 2007, 10.1145/1275986.1275990.

[5] S. Park and S.-Y. Ohm, “An efficient and advanced space-management technique for flash memory using reallocation blocks,” *IEEE Trans. Consum. Electron.*, vol. CE-54, no. 2, pp. 631–638, May 2008, 10.1109/TCE.2008.4560140.

[6] K. Munegowda, G. T. Raju, and V. M. Raju, “Directory compaction techniques for space optimizations in ExFAT and FAT file systems for embedded storage devices,” *Int. J. Computer Science Issues*, vol. 11, issue 1, no. 2, pp. 144–150, Jan 2014.

[7] J. Hwang and K. Won, “A caching mechanism for FAT file system in low-performance embedded system,” *in Proc. ICACT*, Pyeongchang, South Korea, 2016, pp. 799–801, 10.1109/ICACT.2016.7423564.

[8] M. Choi, H. Park and J. Jeon, “Design and implementation of a FAT file system for reduced cluster switching overhead,” *in Proc. MUE2008*, Busan, South Korea, 2008, pp. 355–360, 10.1109/MUE.2008.42.

[9] Winbond W25Q80DV [Online]. Available: www.winbond.com/resource-files/w25q80dv_revf_02112015.pdf, Accessed on: Jun. 4, 2018

Andriy Boubriak was born in Kiev, Ukraine in 1994. He received the MEng in electronic engineering with nanotechnology at University College London, UK in 2016. While at university he interned at Transense Technologies PLC. After graduating he then worked as a design engineer with Zano Controls Ltd. He is currently a hardware and an embedded software engineer with Redwave Labs, Harwell, UK.

Mr Boubriak won the award for Best Final Year Project within the EEE department at University College London for his Masters project on forward error correction implemented with a quantum annealer.

Adam Cooper was born in Barnsley, UK in 1992. He received the BSc degree in music, multimedia and electronics and the MSc degree in embedded systems engineering from the University of Leeds, U.K., in 2014 and 2015, respectively. After graduating he then worked as a Firmware Engineer with Redwave Labs Ltd. He is currently working as a Technology Graduate at YOOX Net-a-Porter Group, London, UK. His current research interests are in front-end web development and progressive web applications.

Christopher Hossack was born near Kingston Upon Thames, UK, in 1969. He received the BEng degree in electronic engineering from the University of Reading in 1990, and the Ph.D. degree in fault tolerance from the University of Reading, in 1994. He was been working on embedded devices for various large companies.

Dmitri Permogorov was born in St. Petersburg, Russia in 1967. He received the MSc from the St. Petersburg University in 1992 and the PhD from Joseph Fourier University, Grenoble, France in 1996. He has had various academic positions in France and Finland, and worked in R&D departments in several companies in the USA and UK. Currently he is the Director of RedWave Labs based in Harwell, UK. He is involved in various electronics and optoelectronics projects across arrange of industries.

Dr. Permogorov enjoys sailing in his spare time.

R. Simon Sherratt (M'97–SM'02–F'12) was born near Liverpool, UK, in 1969. He received the BEng degree in electronic systems and control engineering from Sheffield City Polytechnic, Sheffield, U.K., in 1992, the MSc degree in data telecommunications and the PhD degree in video signal processing from the University of Salford, Salford, U.K., in 1994 and 1996, respectively. In 1996, he was appointed as a lecturer in electronic engineering in the University of Reading, Reading, U.K., where he is currently a Professor of Biosensors. His research interests include signal processing and personal communications in consumer devices focusing on wearable devices and healthcare.

Prof. Sherratt received the first place IEEE Chester Sall Memorial Award in 2006, the second place in 2016, the third place in 2017 and the third place in 2018. He is a reviewer for the IEEE SENSORS JOURNAL and the Emeritus Editor-in-Chief of the IEEE TRANSACTIONS ON CONSUMER ELECTRONICS.