

# Self-stabilizing Embedded Systems

Torben Weis  
University of Duisburg-Essen  
Bismarckstr. 90  
47057 Duisburg, Germany  
torben.weis@uni-due.de

Arno Wacker  
University of Duisburg-Essen  
Bismarckstr. 90  
47057 Duisburg, Germany  
arno.wacker@uni-due.de

## ABSTRACT

The reliability of embedded systems is under constant pressure from miniaturization and cost savings. At some point miniaturized systems built into articles of daily use will show temporary hardware faults, induced for example by temperature changes, radiation of phones passing by, or simply mechanical stress. Thus, software can no longer safely assume that hardware offers fail-stop semantics. This paper shows how to build a self-stabilizing system that can recover from temporary hardware faults. Our approach is mainly carried out in software and requires only little hardware support. The goal is to deliver a low-cost system that can repair itself in constant time from a very wide range of faults.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Formal methods, Reliability*; D.2.5 [Software Engineering]: Testing and Debugging—*Error handling and recovery*

## General Terms

Algorithms, Design, Reliability

## Keywords

Self-stabilization, fault-tolerance, embedded systems

## 1. INTRODUCTION

Today most software is designed for reliable hardware, e.g. servers, PCs, or mobile devices. This hardware reliability comes at a cost. RAM uses extra bits for error correction codes and requires constant power supply and refresh cycles to work properly. CPUs are very expensive because only perfect samples that can operate under various environmental conditions (especially heat) are shipped. Furthermore, the entire computing system must be protected against mechanical stress and radiation. Even a mobile phone passing

by can potentially induce a current leading to wrong computations. While engineering is able to cope with all these, it inflicts costs and hinders further miniaturization.

Another crucial factor is the administration and repair of embedded systems. With further miniaturization and cost reduction the number of deployed devices will increase significantly. This implies that most people will not even be aware of the devices surrounding them. Thus, if the devices do not function correctly, users have no means to manually reset them, simply because they do not even know where the devices are. Finally, such miniature devices cannot feature a reset button since the button would end up being larger than the device itself.

We are facing an environment full of embedded networked devices that are too small and too cheap to offer the reliability we expect from today's computing devices [3]. At the same time they are too ubiquitous and too many to be manually administrated. Consequently, we must develop the systems in such a way that they can recover themselves from faults.

One technique in this area is self-organization. If some devices break (i.e. they detected a fault and turn themselves off), the remaining devices can reorganize to complete their task in a new setting. The same mechanism can help to integrate new devices in the system. However, self-organization usually assumes fail-stop semantics, i.e. a device is either working correctly or it does not work at all. For permanent faults this is reasonable, e.g. if the hardware is physically broken or energy has depleted.

Mechanical stress, temperature changes, and radiation can cause temporary faults. These faults can switch bits in RAM, CPU registers, or inside the CPU's ALU. Ultimately all data stored in volatile memory can be compromised and the device can perform wrong computations. Eventually the fault is over, for example because the temperature is back to normal, or mechanical stress is relieved, or a source of radiation (e.g. a mobile device) has been moved away. At this point the hardware operates normally again, but the device is in an undefined state. At this point, self-stabilization is required, because it has the property of returning the system from any state into a stable state in constant time. Thus, the system can recover from temporary faults.

The paper is structured as follows. In the following section we highlight the key differences between fault tolerance and self-stabilization. Then we discuss in Section 3 a stack of self-stabilizing layers that support the construction of self-stabilizing applications on networked embedded systems. Section 4 contains the main contribution of this paper,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OC'11, June 18, 2011, Karlsruhe, Germany.

Copyright 2011 ACM 978-1-4503-0736-9/11/06 ...\$10.00.

as it shows how to ensure a self-stabilizing program execution on unreliable low-cost hardware. Section 5 concludes the paper and provides outlook on further research.

## 2. SELF-STABILIZATION VERSUS FAULT TOLERANCE

To illustrate the difference between fault-tolerance and self-stabilization it is helpful to divided the states of a system in three classes (which are marked with different colors in Fig. 1):

- Valid states (green)
- Detected invalid states (yellow)
- Undetected invalid states (orange)

If no faults occur and if the application is implemented correctly, the system is initialized in a valid state and then proceeds by moving through a sequence of valid states. A fault can move the system into an invalid state as shown by the red states in Fig. 1a. If neither fault-tolerance nor self-stabilization is realized, the behavior of the system is undefined from there on. Since the system itself is not aware of the fact that its state is invalid, the fault remains undetected. In the best case the system crashes and resets itself. In the worst case it iterates endlessly through invalid states showing undefined behavior.

The classic remedy is to implement fault-tolerance, i.e. the system shows no failure despite a fault – at least for some classes of faults. For this purpose the system must detect that it entered an invalid state. The general problem with fault-tolerance is that fault detection is expensive and works only for a subset of invalid states. For example to cope with an ALU that does not compute correctly, a redundant ALU is required which makes the chip more costly. In the case that both ALUs are subject to the same fault, the redundancy cannot even detect the fault. The additional cost of redundancy contradicts the goal underlying our research, e.g. further miniaturization and cost reduction. Fig. 1b shows how a fault-tolerant system deals with a detected fault. If the fault results in a detectable invalid state (yellow), the fault detectors can safely return the system to a valid state (green). However, not all invalid states are detectable (red). If the system enters such an undetectable invalid state, the behavior of the system remains undefined. In the best case it crashes, in the worst case it loops forever through invalid states.

The concept of self-stabilization has been introduced by Dijkstra [1] and has been subject to further research in recent years [2, 4, 5, 7]. A self-stabilizing system is guaranteed to return from any state (including undetectable invalid states) into a valid state in constant time. Dijkstra illustrated this with a distributed system that passes around a token. A fault can either delete the token or inject duplicate tokens. Nevertheless, the system automatically returns to a valid state where there is exactly one token.

Thus, self-stabilization can guarantee a repair for all states. In contrast, fault tolerance guarantees a repair only for a subset of all states. This powerful feature of self-stabilization comes at a cost: the system does not necessarily detect that it is in an invalid state. Thus, after a fault there is a constant time frame (the stabilization time) in which the system



Figure 2: Stack of self-stabilizing layers

experiences a failure, because it is in an undetected invalid state and its behavior is undefined. After the stabilization time, the system shows no failure any more.

Hence, self-stabilization should ideally be paired with fault-tolerance as shown in Fig. 1c. If a certain kind of fault is easy to detect, fault-tolerance is preferable because it shows no failure despite the fault. Self-stabilization is the final rescue mechanism that catches all undetected invalid states at the cost of a temporary failure.

## 3. HARDWARE & SOFTWARE STACK

Self-stabilization is usually a property of a data structure and the algorithms working on it. Building a self-stabilizing and self-organizing system from ground up requires an algorithm stack as shown in Fig. 2. Each layer in this stack must exhibit the self-stabilization property. This follows from the fact that upon a fault a self-stabilizing layer can still react with a failure for a constant time. The failure of a lower layer is a fault for the upper layer. Thus, the upper layer must in turn be self-stabilizing to recover from the faults induced by the layer below. Hence, it is not possible to handle faults at the lowest layer and shield all other layers from them. This is a major difference to a stack of fault-tolerant layers where lower layers mask a fault where possible. Self-stabilizing systems do not mask faults. They only guarantee a repair.

To stabilize an entire stack, the stabilization must work itself upwards through the layers. Once the lowest layer is stabilized it generates no more failures. Hence, the upper layer experiences no faults any more and can now stabilize as well. This continues until the top most layer is stabilized, too. For this to work it is crucial that failures of upper layers cannot produce faults in lower layers. Otherwise the layers could fault each other in an endless loop and the system would never stabilize. In the following we will quickly iterate over the layers we have developed together with our partners in the MODOC project [6]. The focus of this paper and its novel contribution is the program execution layer that can stabilize from temporary hardware faults. All other layers are only quickly discussed. More detail can be found in [7].

### 3.1 Unreliable Hardware

The hardware is a simple embedded system with constrained memory and energy. Its main purpose is to read sensor data from its sensing components, perform some computation and to control actuators. The hardware has radio and thus can communicate with other devices. If not treated otherwise, we handle the radio as a combination of sensor

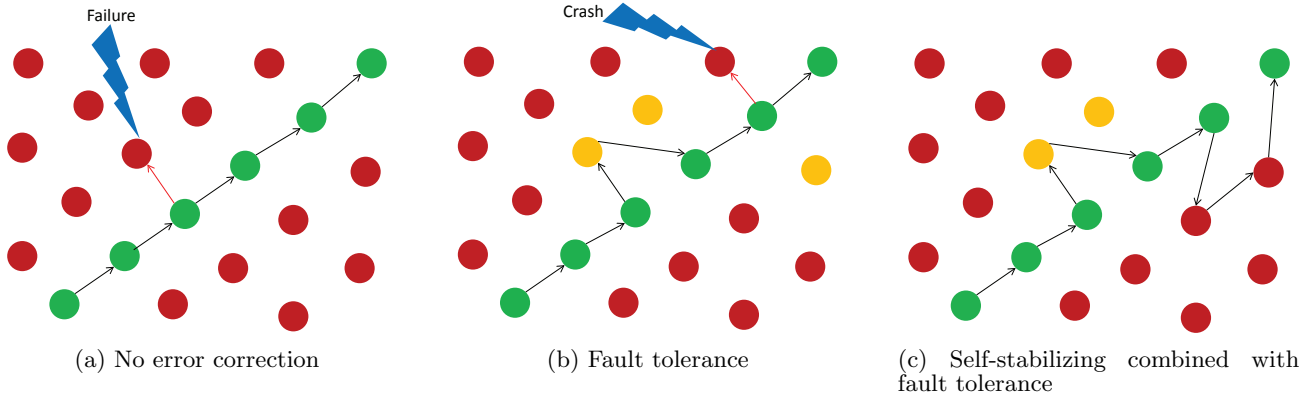


Figure 1: State transitions in programs

and actuator, e.g. receiving messages is sensing and sending messages is like controlling an actuator.

All data stored in volatile memory is subject to faults, e.g. RAM, registers and internal caches. All faults are either fail-stop or temporary. In this paper we do not handle the case that a device is constantly conducting erroneous computations because its ALU is damaged. However, our approach is fine with a malfunctioning ALU as long as this malfunction is temporary, i.e. eventually it will work correctly again.

### 3.2 Program Execution

The purpose of the program execution is to ensure that after a fault the system will return to correctly executing the program. This layer includes tasks that are usually part of an operating system, e.g. memory management. Thus, this layer could be treated as a very simple self-stabilizing operating system. Since this layer is the main focus of this paper, we discuss it in detail in the following section.

### 3.3 Overlay

The overlay is required when multiple devices communicate over a network. Its main purpose is to structure the network in such a way that the next layer can realize a publish/subscribe system. A typical example for such an overlay is a spanning tree that is constructed in a wireless sensor network. Other topologies like a ring are possible as well. The overlay is self-stabilizing, because it can recover from nodes leaving or joining the network even when messages have been lost, duplicated or modified.

### 3.4 Publish/Subscribe

The publish/subscribe system builds on the overlay and is a means to support self-organization in a distributed system. One requirement is that no device has global knowledge of all other devices, because such systems would not scale. This poses the problem of how to figure out which device is doing what and how to send messages to these devices. A publish/subscribe system allows us to send messages to roles instead of devices. This is depicted in Fig. 3. The numbers 8, 15, 42, 62, 80, etc. refer to different roles, for example “temperature sensing”. For example the node in the lower left corner of Fig. 3 has the role 80. Hence, it subscribes to all messages with the topic “80”. If any node wants to talk to the node fulfilling role 80, it can publish a message of topic “80”. The publish/subscribe system is then able to route the

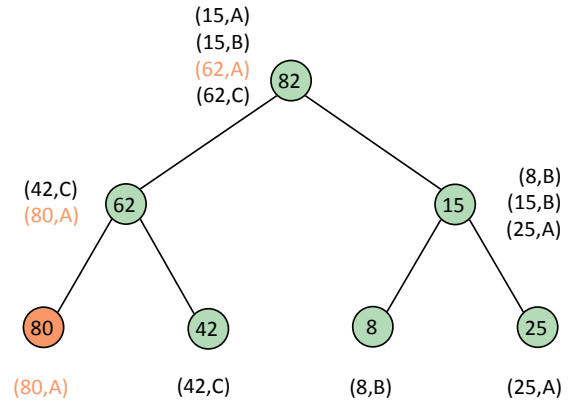


Figure 3: Routing in a publish/subscribe system

message to the node that fulfills this role “80”. It is noteworthy that even the root node has no global knowledge. Each node maintains some state that defines which messages to forward to which direct child in the overlay tree.

The publish/subscribe system can stabilize from corruptions of its routing table. Thus, if faults modify the routing table, routing messages are dropped, duplicated or modified then the routing infrastructure can nevertheless stabilize in constant time.

### 3.5 Role Assignment

The purpose of the role assignment is to guarantee that every node knows which role it has and to ensure that all roles are allocated by at least one node. The role assignment layer uses the publish/subscribe layer to allocate roles and to check which roles are currently taken and which are vacant. The role assignment can stabilize from nodes leaving or joining the network or from erroneous role allocations. Once the publish/subscribe system is stabilized, the role assignment layer can guarantee that all roles are properly allocated again.

### 3.6 Application Layer

The layers discussed so far are application agnostic, i.e. they are useful in many applications of networked embedded systems. The application layer contains the logic that

decides how to control the hardware actuators based on the current system state and the latest sensor readings. The application layer itself must be self-stabilizing, too. Since the most difficult problems are already solved at the lower layers, application development is much easier to achieve than without our layered architecture.

## 4. SELF-STABILIZING PROGRAM EXECUTION

The purpose of the self-stabilizing program execution is to guarantee that

- the program code is executed correctly, i.e. the system remains live
- the memory management stabilizes, i.e. wrong data is either removed from the system or not read anymore

Thus, the program execution layer realizes a minimal operating system. Since embedded systems are usually rather simplistic, the operating system layer can be very thin and thus concentrate on executing the application and managing the memory.

Our system is designed in such a way that the code execution stabilizes first, i.e. after a constant time we can be sure that the original program is executed again. Once that happened, the memory management can stabilize as well. This guarantees that after a constant time the program works only on consistent data. In general we call data consistent when it fulfills two constraints

- **Internal consistency:** All data structures in the memory are structured in accordance with the software specification, e.g. all heap memory is either allocated or free and allocated blocks do not overlap.
- **External consistency:** The data is consistent with the information the system obtained from its sensors, e.g. sensors report that it is dark outside and hence the lights are turned on.

We assume that the executed program can be treated as a function  $F(s_t, s_{t+1}, \dots, s_{t+k})$ , where the  $s_t$  are sensor readings at time  $t$ . The output of the function is a tuple of values that controls actuators, i.e. it turns a light on, or determines the speed of a servomotor. This assumption is a limitation to the kind of applications we can realize. However, for embedded systems it is reasonable to assume that all actions are dependent on the near past and independent of the distant past. By setting the self-stabilization time to  $k$ , it follows that at time  $t$  the machine must not be influenced by any data it received before time  $t - k$ . This is a major difference to the general usage of self-stabilization, which does not require the system to be consistent with sensor readings (external consistency) and thus only requires internal consistency.

### 4.1 Hardware Architecture

To understand how code execution and memory management can recover from faults it is crucial to enumerate the components that may be affected by faults. We assume that the program code is stored in a ROM and this ROM is not subject to faults. This is a reasonable assumption since otherwise the original program is damaged and there is no way



Figure 4: Structure of a machine instruction. The data bytes are optional.

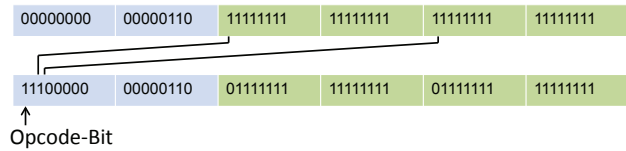


Figure 5: Machine instruction that allows the ALU to detect an erroneous PC

to reconstruct it. However, a temporary fault may include that reading the ROM leads to erroneous results.

We assume that the machine is a register machine with a program counter (PC), a stack pointer (SP) and several general purpose registers. In addition the ALU can contain hidden registers and flags. A fault can alter the value of those registers and flags. The ALU interprets an opcode, reads from memory and writes to the RAM. During a fault, the ALU can compute arbitrarily wrong results and overwrite the RAM in any possible way. The RAM itself may be subject to temporary bit failures. However, after some time we assume that the RAM works reliable again.

Finally, we assume that the system features a hardware clock. A fault can modify the current time or it may cause the clock to run slower or faster during the fault. After a fault the clock must proceed at normal speed, however, the clock time can remain wrong as a result of the fault. Furthermore, the clock can act as a watch-dog, i.e. it must be triggered regularly, otherwise the watch-dog resets the entire system.

### 4.2 PC-Register Repair

The PC register points to the next machine instruction which can be decomposed into an opcode and optional data bytes. Our system assumes that opcodes are always 16-bit large and the entire machine instruction has an even number of bytes. Thus, the machine does not need the lowest bit of the PC Register. This way, the PC can either point to an opcode, to the first, or the third data-byte (see Fig. 4).

If the PC points to a data byte, the ALU will interpret the data bytes as an opcode. Usually this will sooner or later end up in an illegal opcode. However, our system must ensure that an erroneous PC is repaired in constant time. Thus we must mark the data bytes in a special way (see Fig. 4 and Fig. 5). The simple solution is to set the highest bit of *Opcode0* always to one and the highest bit of *Data0* and *Data2* to zero (see Fig. 5). This way the PC must always point to a byte at an even address where the highest bit is set to one, otherwise the machine resets itself. This reset mechanism must be built into the ALU.

This trick means that the four data bytes can contain up to 30 bits. The other two bits are stuffed into the opcode to allow for 32 bits of data per machine instruction. Thus, there are 13 bits remaining to encode the opcode. This simple mechanism assures that after a temporary fault the PC either points to a valid machine instruction or the machine resets itself. While our system design is based on a 32-bit

architecture, the same principles outlined here could be applied to smaller 16-bit architectures.

Finally, the PC cannot address RAM, it can only point to the ROM that contains the program code. If the program code does not fill the entire space in the ROM then RESET machine code instructions are used to fill the remaining space. Thus, if the PC points to a position in the ROM outside the program, the machine resets itself. It is important to note that the PC can still point to any possible machine instruction after a fault. Thus, additional mechanisms are required to ensure a correct program execution.

### 4.3 Recovering from Endless Loops and Repair of Registers

Programs for embedded systems usually feature a main loop or event loop where they wait for new sensor readings. When new data is available, the data is read and processed, output is generated and the program returns to the main loop. Thus, written in pseudo-C the main loop has the form:

```
void main() {
    while( true ) {
        ev = wait_for_event();
        process_event(ev);
        send_output();
        reset_watchdog();
    }
}
```

It is a standard technique to rescue the system from an infinite loop via the watchdog. If it is not triggered regularly, it will reset the entire system. Thus, we can guarantee that after a constant time the system must return to the main loop where it will process new sensor data.

For our system it is important that the stack is empty in the main loop. This way we do not have to worry about wrong data on the stack, because we can safely reset the stack whenever the machine executes `reset_watchdog()`. To reset the stack it is sufficient to put the SP-register to the end of the RAM. In addition, the machine must reset internal flags that can be used by conditional jump opcodes, e.g. the N-bit and the Z-bit. All other registers are reset to zero as well.

Until now, our system guarantees that once a temporary fault is over, it returns to the main loop and resets the stack, no matter how RAM, PC, SP, general purpose registers and internal ALU registers have been compromised. Furthermore, we can be sure that all registers are in their initial state. The remaining problem is the heap, because it is not regularly reset and its data can be altered by a fault resulting in inconsistencies.

### 4.4 Heap-Repair

Repairing the heap can be very easy when a static heap layout is assumed. We show a solution for this case and a more advanced solution that allows for dynamic heap layout.

#### 4.4.1 Static Heap Layout

As stated before all supported applications can be treated as a function  $F(s_t, s_{t+1}, \dots, s_{t+k})$  where the  $s_t$  are sensor readings. For the static heap layout we assume that sensor readings arrive at a fixed rate. Since sensors must be sampled at some fixed rate anyway, this assumption is not unreasonable. By organizing the heap as a ring buffer we can write each new sensor reading at the head of the ring

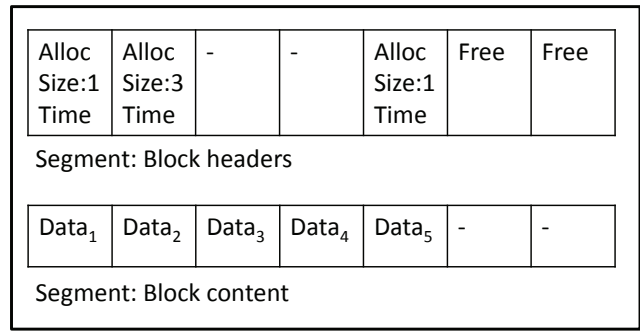


Figure 6: Block layout of the heap. Block headers and block contents are split in two segments.

buffer. The size of the ring buffer is determined by the amount of sensor readings during the self-stabilization time  $k$ , i.e. it must be large enough to store all readings arriving in this time frame. A larger ring buffer does not make any sense, because it would store values which are more than  $k$  seconds old and these must not be used anyway. The first four bytes of RAM are used as an index in the ring buffer that points to the head of the buffer. There is no need to deal with a tail pointer in the ring buffer, because after time  $k$  the ring buffer is always full and then the head and tail are equal anyway. Omitting the tail pointer simplifies the self-stabilization.

We just have to handle the case that the index pointing to the head of the ring buffer is wrong. If it points to the wrong position inside the ring buffer, the system needs  $k$  seconds to overwrite the entire buffer with new sensor readings, thus creating external consistency in the ring buffer. If the head index points outside the ring buffer (e.g. inside the stack), it is helpful to access the memory in segments, i.e. stack and ring buffer exist in two different memory segments. If the index points outside the ring buffer all read/write attempts will violate the segment boundaries and reset the machine. The segment table itself must be written in ROM otherwise it could be compromised by a fault.

The approach laid out above has some shortcomings. First of all, it assumes that the function  $F$  is evaluated periodically. This wastes energy since the machine is always re-computing its output no matter whether the sensor reading did change when compared to previous readings. Second, there is no way to store intermediary results obtained by computation of previous sensor readings. Thus, for every new sensor reading the function  $F(s_t, s_{t+1}, \dots, s_{t+k})$  must be recomputed. Using dynamic heap structures it is possible to store intermediates, i.e. results obtained by computing  $F(s_x, s_{x+1}, \dots, s_y), x \geq t, y < t + k$ , that can be used for computing  $F(s_t, s_{t+1}, \dots, s_{t+k})$  more efficiently.

#### 4.4.2 Dynamic Heap Layout

To allow for storing intermediate results, we need a memory management that can implement the semantics known from POSIX functions `malloc()` and `free()`. Hence, we need to cope with two challenges:

1. The data structures for memory allocation must be self-stabilizing, otherwise the memory management cannot recover from faults that overwrite the RAM.

2. The system must track the “age” of data, e.g. “old” data must not be used any more, otherwise erroneous old data can forever influence the system.

The complexity of the first challenge depends on the efficiency of the memory management. There are three efficiency indicators to consider:

1. The amount of external fragmentation
2. The amount of internal fragmentation
3. The time complexity for allocating and freeing memory

In our approach the heap is split into two segments (see Fig. 6). The first segment contains the allocation headers of all blocks. These headers tell us which blocks are allocated and which are free. Furthermore, the header contains a timestamp for each block. We will discuss timestamps later in this section. The second (and largest) segment contains the contents of the blocks. Each block has a fixed size. Thus we know how many block headers are required.

The size of an allocation is always a multiple of the block size. Thus, our approach has an internal fragmentation in the magnitude of 50% of the block size when we assume that the size of malloc requests is evenly distributed. If the average malloc size is not evenly distributed and well aligned with the block size, our system performs significantly better. Since embedded systems do usually not feature virtual memory, our `malloc()` must allocate consecutive blocks in the physical memory to satisfy large malloc requests. Our current solution does not include heap compaction techniques. Thus, our current approach may suffer from external fragmentation. We intend to improve on this in the future. Our heap structure is ultimately an array of block headers. Statistically we need to traverse half of the array to find free memory or to deallocate memory.

The second challenge (as mentioned above) is to track the “age” of data stored in the allocated blocks. We must avoid that old data stays forever, because it has perhaps been altered by an undetected fault. For this purpose we store a timestamp for each block (see Fig. 6). This timestamp belongs to the block header. The timestamp is the date of the oldest data that has been read before the block has been written. This avoids that old data can be refreshed, because after reading old data the system can only write old data. Thus, if data  $X$  is functionally dependent on data  $Y$ , then  $time(X) \leq time(Y)$ . See [8] for a more detailed discussion of this technique.

To track the time of the oldest data that has been read, we reserve a register in the CPU called “AR” for age-register. Each time data of some block is read, the corresponding timestamp must be read as well. The AR then holds the minimum of its current value and this timestamp. When data is written to a block, the timestamp of the block is updated in the same way. The only source of “fresh” data are sensor readings. Thus, reading from the I/O ports does not alter the AR. Whenever the application returns to the main loop and starts processing new sensor data, the AR can be set to the current clock time, because everything (except the heap) is empty or in the initial state and sensor data is by definition “fresh”.

From an energy perspective it is prohibitive to update the timestamps and AR register upon every read and write, because it requires an additional RAM access each time. However, the advantage is that this can be realized in hardware,

leading to the old concept of tagged memory. We follow an approach where the compiler emits code for updating the AR and timestamps. It is very likely that a loop performs many subsequent reads or writes on one block. Here the compiler could update the AR only once for the entire loop instead of once for each iteration. We have not yet implemented a compiler with these optimizations, but we expect that we can reduce the number of RAM accesses when following this approach. On the negative side, the program size becomes larger because more CPU instructions must be executed to update the AR and the timestamps. Further research must show which solution is best with respect to energy consumption and speed.

So far we have shown that we can build a system that does not deduce fresh data from old data. Now we must discuss the stability of this approach in the face of faults. These faults can change the clock, the AR register and the RAM in arbitrary ways. A modified AR register is recovered after a fixed time, because the system is guaranteed to hit the main loop after a fixed time and there the AR register is set to the current clock time. Furthermore, in the main loop the system checks the heap structure for internal consistency. Therefore, it iterates over the heap structure and checks that the block headers are well-formed and cover all blocks. Upon a violation, the system resets itself. Furthermore, the timestamp of each block is checked. If the timestamp is more than  $k$  seconds older than the current clock (where  $k$  is the stabilization time), the system resets itself, because the application failed to release the memory in time. Assuming that the application is implemented correctly, an outdated timestamp indicates a fault that affected the clock, the AR or the RAM. If the timestamp is in the future, the system resets itself, too. If the clock is modified by a fault, the result is the same as if all timestamps have been modified. Thus, there is no special treatment required. In the end, this guarantees that the heap headers are repaired after a fixed time and all allocated memory blocks contain data that is not older than  $k$  seconds.

Finally, the contents of an allocated block may have been altered by a fault. Often the application logic will detect this. But even if this fault is not detected, we can be sure that the block is deallocated after  $k$  seconds. Thus, if its contents is malformed, the problem is gone after a fixed amount of time. This implies that the heap headers are repaired before we can be sure that the contents of the blocks is repaired. We must avoid that errors in the block contents damage the heap headers, otherwise the system does not stabilize. To avoid that dangling pointers cause the block headers to be overwritten, we place data blocks in a different segment. Thus, malformed block contents may cause more damage in other block content, but it cannot cause damages in the block header.

In summary, the management of a dynamic heap structure is the most complex technique presented in this paper. This complexity comes at a cost, because in regular intervals the system must check the consistency of its heap structure. The simple ring buffer approach does not require this, but it has other shortcomings. Further research must show the trade-off between the different approaches, e.g. does the dynamic heap structure cost less energy than it saves by storing intermediate computation results? Currently, our research focuses on developing several memory management

techniques. An evaluation with respect to energy consumption will happen in a later stage.

## 5. CONCLUSIONS AND OUTLOOK

We have presented a layered architecture for building self-stabilizing embedded systems. The focus of our approach is the support of low-cost miniaturized hardware that can perform self-organization and is able to heal itself from a wide variety of faults. We argued that fault-tolerance alone is no solution because it covers only a subset of faults and often induces redundancy and checking that is costly in terms of money, energy, and space. A special focus of this paper is on the program execution on unreliable hardware. We have shown how to return a system to correct program execution even in the face of faults such as arbitrary changes to the RAM, CPU registers and the ALU itself.

We discussed several options to achieve a self-stabilizing memory management. Our next steps are to compare these options with respect to their energy footprint and their speed. For this purpose we will define a machine language for our CPU and a corresponding interpreter. By counting the executed machine instructions we can estimate how much energy the different algorithms consume.

Another topic of ongoing research is the support of application development. The layered architecture already hides some of the complexity. A specialized programming language or modeling language will further ease the development of self-stabilizing embedded systems. We are currently building a graphical language based on data-flow diagrams which allows to construct self-stabilizing control applications very easily.

## 6. REFERENCES

- [1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17:643–644, November 1974.
- [2] S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, USA, 2000.
- [3] International Technology Roadmap for Semiconductors. *International Technology Roadmap for Semiconductors, 2009 Edition, Executive Summary*, [http://www.itrs.net/Links/2009ITRS/2009Chapters\\_2009Tables/2009\\_ExecSum.pdf](http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf). ITRS, 2009.
- [4] M. A. Jaeger, G. Mühl, M. Werner, and H. Parzyjegla. Reconfiguring self-stabilizing publish/subscribe systems. In R. State, S. van der Meer, D. O’Sullivan, and T. Pfeifer, editors, *Proceedings of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2006)*, volume 4269 of *Lecture Notes in Computer Science*, pages 233–238, Berlin/Heidelberg, Germany, Oct. 2006. Springer.
- [5] M. A. Jaeger, G. Mühl, M. Werner, H. Parzyjegla, and H.-U. Heiss. Algorithms for reconfiguring self-stabilizing publish/subscribe systems. In Mahr and Sheng, editors, *Autonomous Systems – Self-Organisation, Management, and Control*. Springer, oct 2008.
- [6] MODOC - Model-Driven Development of Self-Organizing Control Applications (DFG SPP 1183). <http://kbs.cs.tu-berlin.de/projects/modoc.htm>, 2004.
- [7] T. Weis, H. Parzyjegla, M. A. Jaeger, and G. Mühl. Self-organizing and self-stabilizing role assignment in sensor/actuator networks. In *OTM Conferences (2)*, pages 1807–1824, 2006.
- [8] T. Weis and A. Wacker. Self-stabilizing automata. In M. Hinchey, A. Pagnoni, F. Rammig, and H. Schmeck, editors, *Biologically-Inspired Collaborative Computing*, volume 268 of *IFIP International Federation for Information Processing*, pages 59–69. Springer Boston, 2008. 10.1007/978-0-387-09655-1\_6.