ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Swarm construction: Development of Remote Monitoring Software for Intelligent Structures

Enhancing the Smartblocks with Multitasking

Mémoire présenté en vue de l'obtention du diplôme
d'ingénieur civil en informatique, à finalité spécialisée

Romane Schelkens

Directeur
Prof. Marco Dorigo

Co-Promoteur
Dr. Michael Allwright

Service
IRIDIA

Année académique
2017 - 2018

Abstract

English Version

Smartblocks are dynamically reconfigurable blocks used for autonomous construction applications. They are capable of communicating with each other over a near field communication (NFC) wireless interface. In order to facilitate the development of the system, we envision to develop a software to allow the remote monitoring and controlling the smartblocks when they are assembled into a structure. As it is preferable to connect only one block to the remote computer, other blocks will communicate with the PC by routing messages through the NFC interfaces. However, with the current implementation, the smartblocks can achieve only one NFC exchange at a time. This would result in the information received by the remote computer to be consequently delayed. Therefore, the goal of this master thesis is to provide the smartblocks with multitasking, in order to allow multiple the blocks to communicate with each other simultaneously. Doing so will enable the creation of a remote monitoring system, but also enhance the efficiency of the construction process.

Version Française

Les Smartblocks sont des blocs dynamiquement reconfigurables utilisés pour des applications de construction autonomes. Ils sont capables de communiquer entre eux via une interface sans fil de communication en champ proche (NFC). Afin de faciliter le développement du système, nous envisageons de développer un logiciel permettant la surveillance et le contrôle à distance des smartblocks lorsqu'ils sont assemblés en une structure. Puisqu'il est préférable de ne connecter qu'un seul bloc à l'ordinateur distant, les autres blocs communiqueront avec le PC en acheminant les messages via les interfaces NFC. Cependant, avec l'implémentation actuelle, les smartblocks ne peuvent réaliser qu'un seul échange NFC à la fois. Cela aurait pour conséquence que les informations reçues par l'ordinateur distant soient par conséquent retardées. Par conséquent, le but de ce mémoire est de fournir aux smartblocks la capacité de faire du multitâche, afin de permettre aux blocs de communiquer entre eux simultanément. Cela permettra la création d'un système de surveillance à distance, mais aussi permettra d'améliorer l'efficacité du processus de construction.

Acknowledgements

I would first like to thank all the people that dedicate themseleves to the development of free and open-source softwares. Without them, my master thesis project would have been impossible. But more importantly, they give hope for a brighter future, as they demonstrate the powerfulness of free knowledge and large-scale collaboration.

I would like to also express gratitude to Michael Allwright and Marco Dorigo for their help and their patience.

I am grateful to Margaux for listening to me everytime I needed to clarify my ideas, explain the obstacles I was facing and think aloud the possible solutions. Her company and kindness have been of a great help. Thanks also to Eleonora, Tina and the house, for their company, their hospitality and the good meals.

I would like to thank my family for their support, in particular my parents or offering these studies that I like, and Hortense for brighting up our days with her beautiful smile. Finally, thanks to Malcolm for his constant presence, support and advise, and for all the joy and good moments he offers me.

Contents

Part 1

Introduction

Autonomous construction is envisioned for construction in remote and hazardous environments, where human presence is impossible, such as deep sea or outer space. When done with swarm robotic - a field of artificial intelligence drawing inspiration from the behaviour of social insects - the process is very robust, efficient and easily scalable. However, even if it has been proved in simulation, not many hardware implementations exist. Using the S-bots created at IRIDIA, ULB, the work of M. Allwright [1] aim to produce a working implementation of autonomous construction system with swarm robotic. They are shown in figure 1.1.

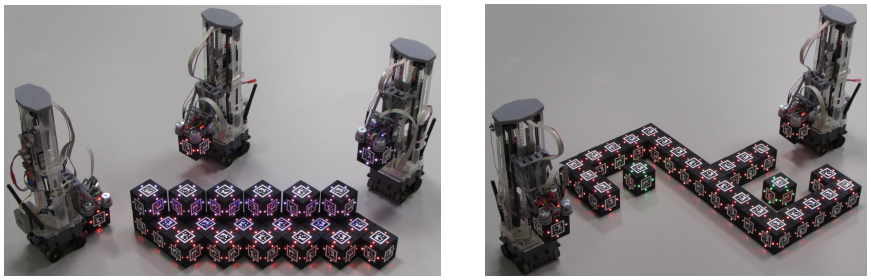


Figure 1.1: The robots of M. Allwright constructing two different structures

Such systems are greatly enhanced by using intelligent building blocks instead of simple bricks. Doing so enhance the efficiency, the robustness and simplifies the requirements for the manipulating robots. In this master thesis, we discuss how and why (Part 2) and review different hardware implementations of intelligent blocks that exist nowadays (Part 3).

We also present in details a software that was made as the practical work of this master thesis. It was made to run on the building blocks created by M. Allwright. These cubic blocks, called stigmergic blocks or smartblocks, are able to compute and store information, learn about their orientation using an accelerometer, to communicate via Near Field Communication (NFC) and to light up LEDs in different colours. They are meant to be manipulated by a swarm of robots with computer vision abilities in order to create three-dimensional structures. The LEDs will serve as stigmergic signals (that we can also call block-to-robot communication), and NFC will be used robot-to-block and

block-to-block communication.

Our goal was to provide the Smartblocks with multitasking. Indeed, the initial smartblock's software is working with a single task, meaning that it is impossible for a block to simultaneously communicate with more than one of its neighbours, or to simultaneously communicate with a neighbour and do something else, like computing information or controlling the colours of the LEDs on one face. For larges structures, where many messages can be passed in every direction, this would cause the routing process to be severely slowed down, especially if for messages that have to travel a big distance before reaching their destination. For example, if a block has to send a message on all of its six faces with a single task, it would have to carry on the communication on one face at a time, thus the transmission on the sixth face will occur only after five times the transmission time.

Such slow communication is problematic when we want the Smartblocks to learn about the global structure via the block-to-block interfaces and change their signals according to it. Indeed, one Smartblock would need at some time to receive a message from a distant other block in order to update its signal. During the time where the message is not yet received, it will cause the smartblock to display obsolete information, thus eventually causing errors in the construction process.

Additionally, this prevents the creation of a fast enough monitoring and controlling software that would connect to only one block and reach all the others by passing messages through the block-to-block interfaces. The information obtained by the distant computer will be obsolete and the control messages sent to one particular block will reach it later than expected.

So multitasking has been envisioned as a solution to reduce this communication delay. Multitasking allows several tasks to be executed concurrently. The tasks are not really executed in parallel, as it is impossible to do so with only one processor. Instead, the apparent concurrency is achieved by rapidly switching between the ongoing tasks. At final, the tasks may be slightly slowed down because they have to share the computing time, but concurrency is achieved. Moreover, it occasionally happens that a task consumes computing time just for waiting, it is called *busy-wait*. With multitasking, this computing time can be reused for another task, thus overall efficiency is enhanced.

The implemented software is explained in details (Part 4), and the results we obtained are presented (Part 5).

Part 2

Context : Intelligent Blocks for Swarm Construction

This section presents the context of our research. Indeed, as we will see in the next section (Section 3), intelligent structures formed by an assembly of communicating blocks can serve several purposes. But in our case, the Smartblocks are meant to be used as the building blocks of a swarm construction system.

As such, it is valuable to present here this field of autonomous construction (Section 2.1) and to explain what are the advantages brought by the use of intelligent blocks, also called semi-active building material (Section 2.2). Additionally, we briefly present what is an active building material (Section 2.3), as we will also encounter those later in the master thesis.

2.1 Swarm Construction

Swarm intelligence is the field of artificial intelligence that focuses on collective behaviour emerging from a swarm of simple independent agents. The agents typically react only to local information or local communication with simple rules and, when carefully designing those rules, one can make them self-organize to collectively solve a desired task. The global behaviour is thus a property emerging from the distributed system, in contrast with a centralized system where the global behaviour is imposed by only one decision makers. The systems and algorithms designed in this field take inspiration from the observation of self-organized systems in nature, like bird flocking, ant colonies, fish schooling, etc. Swarm intelligence has been proved useful for several applications, like optimization, data mining, and robotics.

In particular, the application that interests us here is the autonomous construction of structures by robots. Robotic construction is envisioned to be a solution for working in environments where human presence is impossible or too hazardous. And when remote control is unachievable because a wireless connection cannot reliably reach the construction site or can reach it but with too much latency, thereby preventing real-time reactions to threats, then the construction system needs to be completely autonomous. Mines, deep sea and outer space are some examples.

We talk about swarm construction when the autonomous construction is achieved by a swarm of robots driven by a decentralized control strategy inspired by swarm intelligence principles. The advantages of using swarm construction are:

- **Robustness**, because there is no single point of failure. One robot can fail without compromising the whole system. Also, robots are simpler thus less prone to failure.
- **Scalability**, because it is possible (and eventually cheap) to add one or many more robots on-the-go and without reprogramming.
- **Adaptativity**, because a swarm of robots is typically able to adapt to modifications in the environment.
- **Parallelism**, because different subtasks can be performed by different robots at the same time.

As swarm intelligence, swarm construction also draws inspiration from nature. Indeed, it has been observed that social insects, like ants, bees and termites are able to construct large and complex nests (see Figure 2.1) without any centralized instruction. Their behaviour has been studied, adapted to robotics and enhanced in order to construct many kinds of structures, from the simplest to the most complex ones.

Figure 2.1: (left) A termite cathedral in Northern Territory, Australia; (right) weaver ants Oecophylla smaragdina stitch leaves together to build their nests in Thailand.

One of the first algorithms was proposed by Deneubourg [2] in 1977 and draws inspiration from ants. In its simplest form, it allows to construct clusters, and hence it is also useful in database engineering for clustering tasks. The idea is as follows: The agents randomly walk an area and load or unload items based on the density of items observed at their location. As an example of extension, K. Sugawara and Y. Doi [3] proposed to use a building material capable of computation and communication, and slightly change the condition for unloading items. In simulation, their algorithm allows constructing any 2D structure specified by a user, and is explained more extensively in section 3.1.6.

Another model was proposed by Theraulaz and Bonabeau [4] in 1995. They have proven its ability to build several kinds of complex 3D structure in simulation, as shown in Figure 2.2. In their system, agents search in a 3D lattice for predefined patterns that will trigger the deposit of one item. This newly placed item then eventually modifies the

structure in such a way that a new pattern appears, which in turn will trigger the deposit of a new item, and so on. This control strategy - with adaptations to use intelligent blocks instead of inert ones - is what M. Allwright et al. [1] [5] implemented in their research.

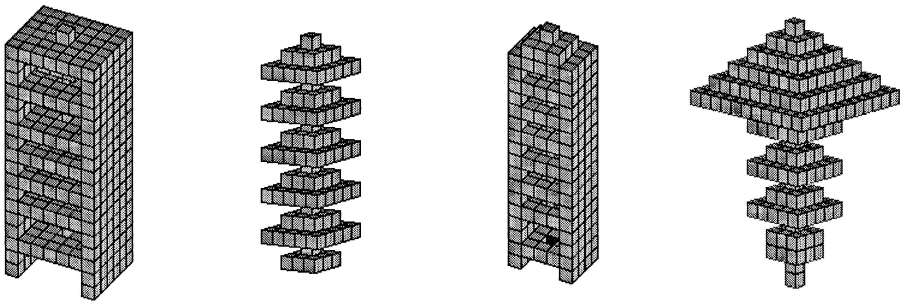


Figure 2.2: Generated structures by Theraulaz and Bonabeau

Swarm construction is guided by a particular type of communication, called *stigmergy*. It was first observed by Grassé during a study on the collective construction behaviour of termites. He defines it as follows: "*The coordination of tasks and the regulation of constructions does not depend directly on the workers, but on the constructions themselves. The worker does not direct his work but is guided by it. It is to this special form of stimulation that we give the name STIGMERGY.*" [6]

Stigmergy can also be defined as the communication achieved by "*storing information in the environment*" [7]. In swarm construction, it means that the agents indirectly communicate through modification of the structure being constructed.

So the building material is the means of communication. As such, choosing a building material will influence the sensing capabilities (computer vision for example) needed for the building robots. The other role of the building material is, of course, to compose the structure. For this, different situations require different degrees of solidity, flexibility, impermeability, etc.

Therefore, a great variety of building materials have been considered for autonomous construction. They can be rigid, like bricks, or non-rigid, like ropes, glue or bags of sand. They can be in many shapes: cubes, spheres, bars, and others. They may be provided or not with mechanical or magnet-based connectors, in order to enhance the robustness of the final structure and to reduce cumulative misalignment. And finally, the building material can be either homogeneous, composed of only one kind of items, either heterogeneous.

Traditionally, materials used for construction are inert, passive. But in the field of autonomous construction, some are given additional abilities. In that respect, one can classify building materials in three categories:

- **Passive** materials have no additional abilities.
- **Semi-active** materials are able to sense, compute, store and communicate information.

- **Active** materials are able to sense, compute, store and communicate information, and in addition, they are capable of motion.

We discuss the two latter in the next two sections.

2.2 Semi-Active Building Materials

Semi-active materials are defined here as blocks able to sense, compute, store and communicate information.

While they present a higher unit cost, they allow - in general - to use simpler building robots. However, the cost of the overall structure will still be more expensive than with passive materials, because at final we need more electronic devices. But on the other hand, they present several advantages that can be determinant.

First, it has been shown that using them significantly improves the robustness and the efficiency of an autonomous construction system that uses swarm robotics. "*Added block capabilities increase the availability of nonlocal structural knowledge, thereby increasing robustness and significantly speeding construction.*[7]

Secondly, the principle of stigmergy as used by social insects can be used to produce structures with given qualitative characteristics [8], but it does not easily allow the consistent production of specific structures [7]. Semi-active building materials can overcome this limitation, because they are able to distributively store and share a high-level definition of the structure, and adapt the signals they send to the manipulating robots according to it. Thus, for certain schemes, enhancing the capabilities of the building material may also extend the class of feasible structures.

So in a sense, the intelligence is distributed amongst both the blocks and the building robots, rather than only over the swarm of building robots. As such, it is important to note that when constructing a structure with such materials, we get a structure that is capable of intelligence, that can be called *intelligent structure*. This fact is another, third advantage.

In the following, we discuss these three advantages with regards to examples we found in three articles:

- *Collective construction of Dynamic Structure Initiated by Semi-active Blocks* by K. Sugawara and Y. Doi [3] (whose hardware is presented in section 3.1.6),
- *Distributed Construction by Mobile Robots with Enhanced Building Blocks* by Werfel et al. [7],
- The work of Allwright et al. [1][5], on which the practical work of this master thesis is based (section 4).

2.2.1 First Advantage: Robustness and Efficiency

In systems using semi-active materials, the responsibility of knowing where a new block is needed is switched from the robots to the building blocks. The robots thus just have to correctly read the signals sent by the blocks and act accordingly. This means that we can eventually use robots with simpler sensing abilities and with less computing power. Simpler robots are less prone to failure, so we gain in robustness. But conversely, additional failure points are brought by the use of intelligent blocks. Not as much though, because the absence of mobility removes a frequent cause of failure.

But swarm robotic system are already very robust per se because, as we know, the system will continue working even if one or several moving robots fail. Until, of course, the utmost situation occurs where all robots are lost, causing the construction process to stop irrevocably. But studying the robustness is also relevant because a decrease in robustness leads to a decrease in efficiency. Indeed, losing one or several robots will slow the construction process down, and eventually drastically.

In the system discussed in [7], the moving robot needs to get at some point the knowledge of where it is located with respect to the structure it is constructing. With indistinguishable passive blocks, it has to first search for the unique "landmark" block, then follow the perimeter of the structure while counting the number of blocks it passes by. Only once it arrives at a location where it knows a block is needed, it can finally deposit the block it was carrying. Additionally, if a robot drifts away from the perimeter, then, after regaining it, it will need to travel all the way back around to the marker to ensure knowing its location correctly. On the other hand, with semi-active blocks, it gets to know its location as soon as it reaches the structure. Moreover, it does not need to learn about the location anymore, thus it must not keep the entire map of the structure in its memory, because the blocks can directly indicate where a new block is needed. So by this, we understand that the method using passive blocks is very slow when compared to the one with semi-active blocks. Moreover, with passive blocks, if the moving robot somehow miscounts the blocks it is passing by, it could cause it to deposit a block at a location that was meant to stay empty. This error will cause further errors and, by this, cause the entire structure to be constructed wrongly. So to conclude, this paper shows that using enhanced blocks greatly increase robustness and efficiency, at least with the algorithm proposed in it.

In M. Allwright's work, moving robots are simplified also by hardware choices. For example, it needs less precision because robots present a limited self-alignment feature. Also, it needs simpler computer vision abilities because QR tags are attached to each block. These are not features proper to semi-active materials, but it is worth mentioning here, as it implies both a gain in speed and in robustness. In speed, because the moving robots will be able to spot patterns a bit faster. In robustness, because if we use simpler computer vision abilities, identification of patterns is less likely to fail. And, as mentioned in the previous paragraph, more failures may cause moving robots to misplace one block and, with chain reaction, many other blocks will be misplaced causing the system to construct a totally wrong structure.

We can observe another cause of robustness and efficiency increase in M. Allwright's system. Indeed, Theraulaz and Bonabeau noticed that no interesting pattern could be generated using a single type of bricks.[9] With intelligent blocks, a robot would be able to change the type of the brick just before placing it on the structure. We thus avoid the need for searching for a certain kind of block in the surroundings, as every block is suitable. This can be a significant gain of time, especially when there are many types of blocks.

2.2.2 Second Advantage: Extension of the Class of Structures

The work of K. Sugawara and Y. Doi [3] is based on Deneubourg's algorithm. This algorithm originally can only construct clusters. By using semi-active materials and slightly change the condition for unloading items, they demonstrated - in simulation - the creation of any 2D structure specified by a user.

Using the semi-active material implemented by M. Allwright also enable a greater variety of structures than using a passive material. As explained in section 2.2.3, semi-active blocks can change the signal they send at any time, and this allows to change the structure on-the-go. With this property, a block placed "green" on the structure may decide to become "red", as opposed to the passive blocks that will always stay "green". Therefore, the next block will be placed differently on the structure containing semi-active materials than on the one containing passive materials. But the "green" block can also be "blue", "yellow", etc. Consequently, with semi-active blocks, we multiply the number of possibilities for the placement of the next block. We thus obviously multiply the number of structures that can be made with the system.

Additionally, there could be another way to construct many different types of structures. The smartblocks of M. Allwright are provided with NFC peer-to-peer communication. We could use it for block-to-robot communication, in the idea that the block commands to the moving robot its next move. Doing so, the "pattern" that the moving robots need to recognize is just one block. The drawback is that they need to be in contact with it to "see" the signal, thus causing additional latency and probably congestion. But on the other hand, we have an infinite possibility for the patterns, while with the scheme of Theraulaz and Bonabeau we had to cope with a limited amount. So this way of functioning leads to infinite possibilities for the structures, but with severe drawbacks.

But of course, this is not always true. Werfel et al. present in [7] an algorithm to construct 2D structures that can be applied with any sort of building materials: passive, passive but uniquely labelled, able to be relabeled by robots, and finally semi-active. The use of semi-active materials here thus doesn't add to the types of structures that can be constructed with their system.

2.2.3 Third Advantage: Production of Intelligent Structures

By using semi-active blocks, we create structures that have communication, computing and sensing abilities. We then talk about *intelligent structures*. Intelligent structures are envisioned to be found in critical infrastructure, principally to prevent accidents. A typical example is a bridge that can sense when there is too much traffic and react by reinforcing or alerting.

For example, K. Sugawara and Y. Doi demonstrated a useful application for semi-active materials. They highlight and use the fact that, if we implement that the robots cannot differentiate between placed and unused blocks, then a structure constructed with their system is in dynamic equilibrium (meaning blocks are constantly moved locally, but the structure globally stays the same). Indeed, by providing the building material with some intelligence and the ability to detect an external stress, the blocks can modify the signals sent to the robots in order to make them add more blocks where a risk is detected. It will serve, for example, to strengthen a wall in order to resist a strong wind or a flood. This behaviour was implemented in simulation, as shown in Figure 2.3.

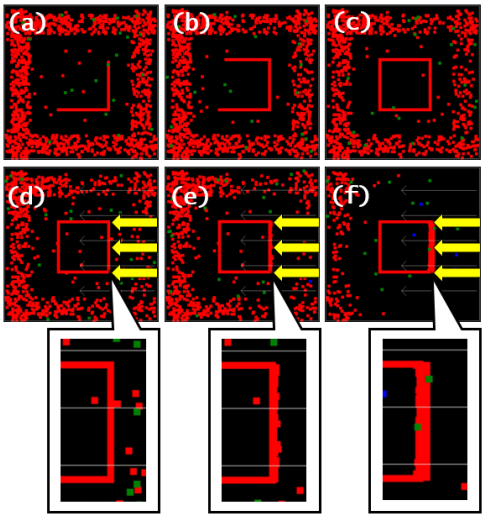


Figure 2.3: The simulation of the reinforcement of the wall, by K. Sugawara and Y. Doi. A square structure is assembled (a-c). Once an external stress is applied (yellow arrows), the blocks on the right side of the structure start to transmit signals and reinforce the wall by increasing its thickness (d-f).

Similiarly, for the system implemented by M. Allwright, we could imagine that a block can change its type in reaction to certain events. By changing the type of a block, we can make a certain pattern appear or disappear. As the actions of the constructing robots are driven by the patterns they observe, it means that we allow the structure to change on-the-go, in reaction to a threat or to a change in the environment. And if, as in the previous paragraph, we allow the robots to also withdraw blocks from the structure, then it becomes an intelligent structure able to reinforce itself against a threat.

However, one limitation is caused by the power consumption of intelligent structures

constructed in this way. First, as many intelligent blocks compute and communicate, the consumption may be higher than for other intelligent structures. Secondly, if the structure is envisioned for a remote area, where no power supply is possible, then the structure has to integrate batteries. Doing so, its existence is limited in time. A solution can be to add internal power generator, probably based on renewable energies, like sunlight or tidal energy.

2.3 Active Materials

Active materials are able to sense, compute, store and communicate information, like semi-active ones, but are enhancing the building blocks a step further by adding the ability to move.

They are meant to rearrange themselves autonomously into structures. In a sense, they are both the component and the constructor. Systems composed of such materials are called *self-assembly systems*. They can also be driven by swarm robotics algorithms.

Self-assembly is strongly associated with research in the fields of programmable matter and of modular robotic. Such systems are more robust to failure because, if one robot malfunction, the others may continue to self-assemble. Also, as they do not rely on other moving robots for placement, they can be more appropriate for several applications, like when space is limited, when robots moving independently are impossible, or when the goal of the construction process is purely artistic (see Figure 2.4 right). Finally, their aesthetic advantage seems to inspire collective imaginary more than for other autonomous construction systems, as shown in Figure 2.4. However, the units are expensive and the cost does not decrease when scaling to larger structures. Plus, the need for coordination between units make such systems highly complex to develop and program.

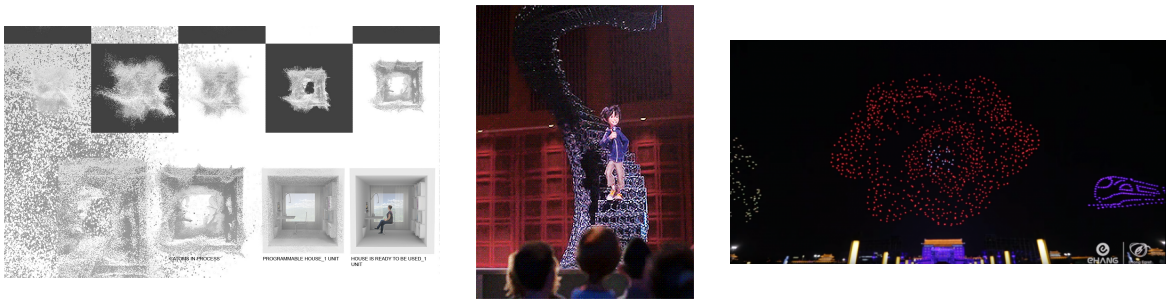


Figure 2.4: (left) A concept of programmable appartment complex, imagined by Monika Rafajova (source: `http://www.studioflorian.com/projekty/252-monika-rafajova-programmable-matter`); (center) An image from the movie *Big Hero 6*, where a stair is composed of a set of small magnetic robots; (right) An exemple of artistic project : a 3D image formed by 1374 drones, by the company *Ehang*, in Xi'an, China, on May 1st, 2018.

Part 3

Related work

Since our work focuses on intelligent blocks, and on their ability to distributively learn about the structure they are forming, it is more tied to the studies concerning intelligent structures - more specifically any modular structures formed by units capable of computation and communication. Autonomous construction is not the only purpose intelligent structures are used for.

In the following (3.1), we briefly present different implementations that exist nowadays and, when possible, we highlight how communication inside the structure was implemented. The Smartblocks we are working on are also included in this review. Then (3.2), we compare them on several characteristics - characteristics that were chosen while keeping in mind our goal of using them in swarm construction systems. Finally (3.3), we conclude by discussing which are suitable for autonomous construction with a swarm of robots and which are not.

3.1 Overview

3.1.1 Building Virtual Structures With Physical Blocks by Anderson et al.

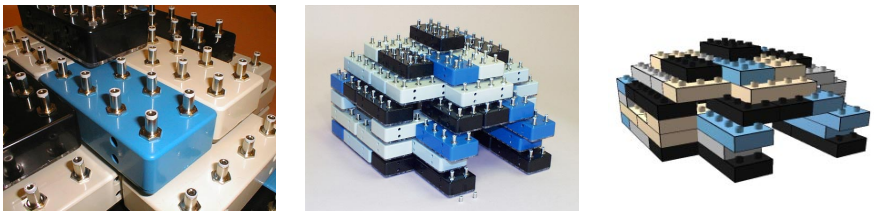


Figure 3.1: A zoom on the blocks created by Anderson et al. (left); A structure made with the blocks (center); The same structure rendered by the software (right).

The goal of this work [10] is to create a tangible interface to create virtual structures. Figure 3.1 shows the blocks and an example of rendering. Here the preview is rendered by sticking to the reality but the software can also apply other textures. The principle

is as follows: a user assemble the blocks into the desired structure, then, when he is satisfied, he connects a special block - called *the drain* - that provides power and has a serial communication link with a computer. This trigger the following series of events to happen in order to render the structure on the computer:

1. Each block determines which of its pins are connected to another block.
2. Each block exchanges block and pin ID with their neighbours thanks to their connected pins.
3. Data is sent from each block to the computer via a distributed depth-first traversal of the structure.
4. The computer renders the structure using the data collected along with previously collected information about each block's colour and shape.

Messages and power are passed from block to block via a modified DC pin, that also serves as a structural link. It should be noted that the blocks only use the power-on event to synchronize. To tackle the fact that synchronization is approximative, messages are passed in a pipelined way. This scheme is interesting because it implements a basic remote monitoring software, where block-to-block communication is used to allow a remote computer to learn about the structure via only one special block. However, remote control, i.e. allowing the computer to send instructions to the blocks, is not implemented.

3.1.2 Boda Blocks by Buechley et al.

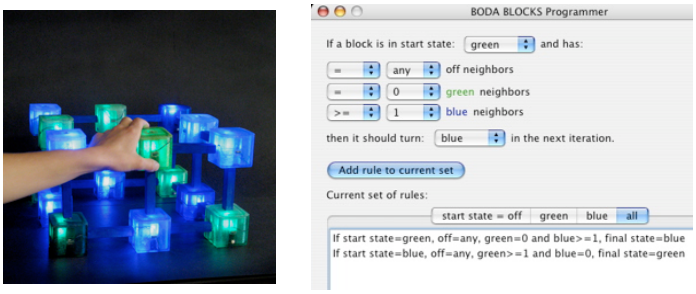


Figure 3.2: A structure made with the Boda blocks (left); The software used to program them (right).

The Boda Blocks are meant to be used for educational purpose [11], to let children explore how local rules can result in complex global patterns. Each block acts as a cellular automaton and displays its state by changing its colour. A light located inside the block displays the colour and allows three states: blue, green or off. When in *executing mode*, the blocks change their state at each timestep, based on their current state and on the state of their neighbours, according to the rules defined by the user. The software used to define those rules is shown in Figure 3.2. Physically, the cubic blocks are connected with a long bar.

Because the Boda Blocks operate with local rules, the block-to-block communication is used only to exchange state information with the neighbouring blocks. There is no routing or message-passing scheme to discuss.

3.1.3 Blinky Blocks by Kirby et al.

Figure 3.3: A structure made with the Blinky Blocks, running the "rainbow program" (left); Details of the blocks's top, bottom and inside (right).

The idea of Blinky Blocks [12] came from the observation that there is currently no programmable matter that is affordable enough. As part of the *Claytronics* project, 100 of these glowing lego-like blocks were manufactured. They are able to sense changes in their orientation and sudden impulses, capture and play sound, and glow in any colour. Attachment (and correct alignment) is ensured by magnetic forces horizontally and lego-like pins vertically. Messages and power are passed from block to block via electric spring pins on each face.

Interestingly, each block's behaviour is not meant to be programmed individually, but rather the global behaviour has to be described by expressing typological conditions with the distributed programming language *Meld*. The program then compiles into programs that will be run on each block. The way messages are passed into the structure is then quite hidden. An example of a program running on a structure made of Blinky Blocks is shown in Figure 3.3.

3.1.4 Robots Pebbles by Gilpin et al.

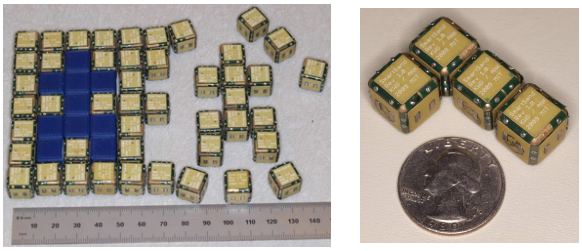


Figure 3.4: An example shape duplication with the Robots Pebbles (left); Zoom on the Pebbles forming a simple 2D structure (right).

The Robots Pebbles [13][14] are an impressing advance towards the miniaturization of intelligent structures. The modules are cubes of 1cm length that uses four electro-permanent magnets to bond, to self-align, to communicate, and to share power with their neighbours. However, the system is currently limited to 2D.

With these, Gilpin et al. developed an algorithm for autonomous shape duplication through self-disassembly. The idea is to submerge an object in a vat of smart particles.

When the user sends a signal, the smart particles magnetically bonds, then collaboratively run the shape duplication algorithm that ends in all the particles detaching except the ones forming a new duplicated shape. Finally, the user can brush the particles aside to reveal both the old and the newly formed object. Currently, the shaking of a bag is replaced by a vibrating table - the equivalent in 2D. An example of shape duplication is shown in Figure 3.4.

Several steps of this algorithms require messages to be propagated or sent to one block in particular. For this, the authors chose the *Bug2* algorithm, which was primarily designed for robot motion planning. Here the moving robot - called "*bug*" - is the message instead. It only needs to access local information: its position and, for all its four faces, whether it is in contact with an obstacle or with another block - two pieces of information computed by the Pebbles at the start of the algorithm. Therefore, the *Bug2* algorithm only requires a fixed amount of memory that can be stored easily, as opposed, for example, to the use of routing tables that needs more memory especially when the system's size grows. Moreover, the way a "*bug*" circumvent an obstacle is well suited for the shape sensing phase where the message has to travel along the perimeter of the shape to duplicate. The algorithm is resistant to missing communication links and adapts to many typologies. The drawback is that it does not function in 3D, but the authors are currently researching a way to divide one 3D problem into several 2D subproblems.

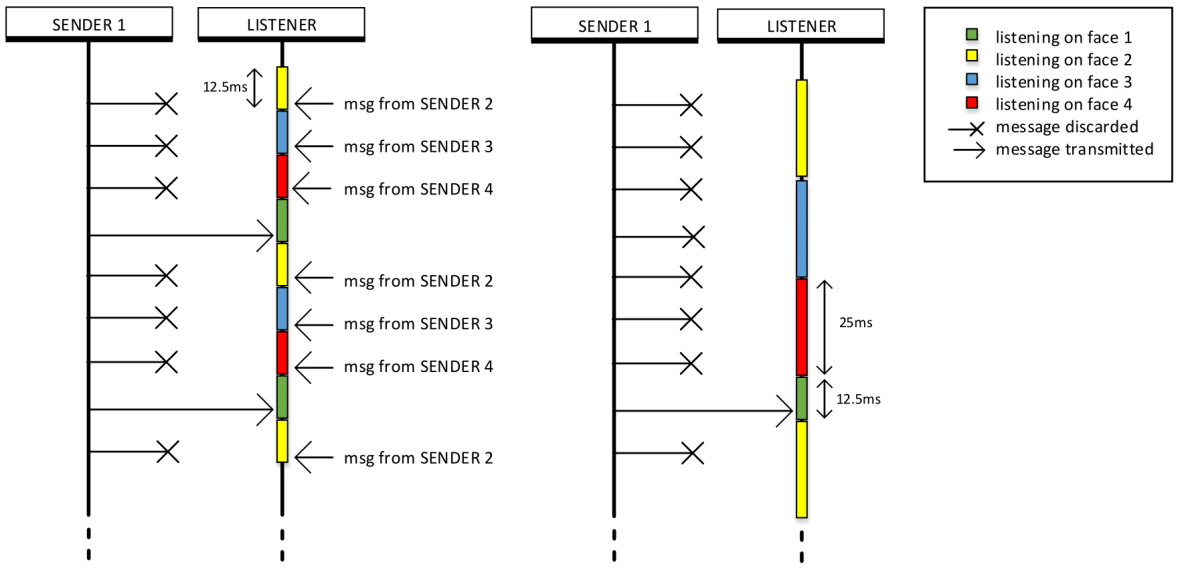


Figure 3.5: Time diagrams showing the communication between two Pebbles when the one receiving the messages is surrounded by four other Pebbles (left) and when it has only one neighbour (right).

Interestingly, in article [13], they highlight how block-to-block communication performance is limited by single-tasking: as the Robot Pebbles does not use multitasking, it has to divide the time listening to messages between its four faces. They made experiments with one robot programmed to listen on each of its four faces, in turn, during 25ms. It is surrounded by one or several other robots continuously trying to send messages to it. It

showed that about 25% of the messages are received, and that the number of successive unsuccessful attempts was rarely more than three. Indeed, during those three attempts, the listening robot is busy listening to other faces. But the percentage of successful transmission (and the message exchange rate) increases when increasing the number of transmitting robots. This is due to the fact that, in their settings, receiving a message takes only 12.5ms, while listening without receiving takes 25ms. Thus, when it receives a message, the listening robot switches more quickly to listening on the next face. Those two situations are illustrated in Figure 3.5.

So this shows that without multitasking, or at least some interrupts driving the communication process, the communication performance is slowed down. This is quite relevant to highlight as communication is a major bottleneck for system performance.

3.1.5 M-Blocks by Romanishin et al.

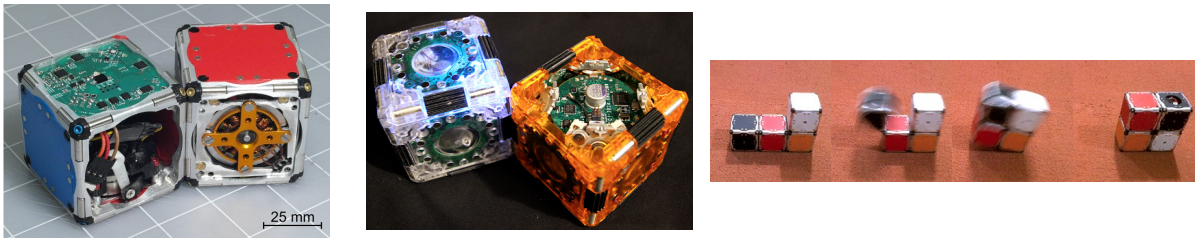


Figure 3.6: Version 1 of the M-blocks, having a fixed flywheel (left); Version 2 of the M-blocks, whose flywheel can change axis, allowing 3D pivoting motion (center); One of the motion the M-blocks are capable of (right).

The M-blocks [15][16] are an implementation of active materials, meaning they are capable of motion. They contain a flywheel that accumulates momentum, and movement is triggered by suddenly stopping this flywheel. Therefore, they are only capable of pivoting motion. The types of movements that the M-blocks can achieve is defined in the "pivoting cube model" (PCM), as opposed to the "sliding-cube model" (SCM), that is more commonly used - but more tricky to implement - in researches about active materials. Furthermore, for 3D M-blocks, the flywheel can pivot inside the block allowing movements in every direction. This works either in a free environment either inside a structure made of M-blocks. In the latter case, self-alignment is helped by magnets. Figure 3.6 shows a M-block changing its position in a structure.

While it is a promising step towards autonomous construction with active materials, for now M-block's movements are only monitored remotely. The first version of M-blocks uses wireless communication through *Xbee* modules. The second uses Bluetooth and has the capability to use the *ant algorithm* for routing. However, no example of use of this routing scheme for a distributive construction process was published yet. For the future, the authors envision to use infrared (IR) communication.

3.1.6 Collective Construction of Dynamic Structures by Sugawara and Doi

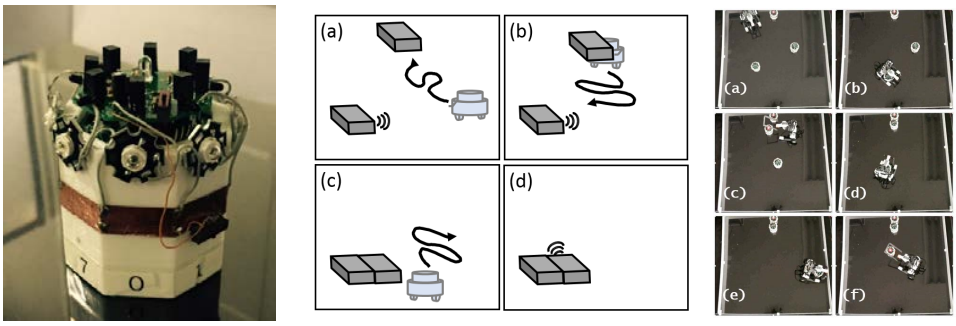


Figure 3.7: Picture of the block created by Sugawara and Doi (left); Schemas showing the functionning of the system (center); Picture showing the manipulating robots constructing a simple line of 3 blocks (right).

The octagonal blocks made by Sugawara and Doi [3] are meant to be used as semi-active building material for autonomous construction of 2D structures with swarm robotic. We can first note that power is internal, and that no attachment method is provided. They communicate with the moving robots with two mediums: one LED on the top indicates if the block is placed as part of a structure or not, and eight infrared LEDs (one on each face) serve to send a per-face signal. Block-to-block communication is implemented using these eight infrared LEDs and eight photosensors. With this, integer values can be passed from one block to another by blinking the infrared LED. The value corresponds to the number of pulses observed during a certain period.

The algorithm driving the manipulating blocks is inspired by Deneubourg's algorithm (see Section 2.1): they randomly walk an area, pick the block if it has its top led turned off, bring it to the structure and attach it where they sense an IR signal. We can see the algorithm in action in Figure 3.7.

The algorithm driving the blocks is defined as follows: the blocks store a counter (a simple integer), representing the length of a line to be made. When a new block is added to the structure, it receives the value via an IR light blinking, and it decrements it by one. Blocks also store and communicate rules for creation of new lines. Basically, a rule state: "When counter reaches value t, initiate a line in direction d and (eventually) set the counter to a new value n.". Therefore, they can be defined by a tuple (t, d, n).

So we see here that blocks use relatively basic hardware, which is an advantage in term of manufacturing cost. Also, the message-passing scheme is kept simple: a message is passed only when a new block get attached to the structure and on no other occasion. The message itself is kept simple: it only consists in a series of integers containing a counter value and eventually several rules, each defined by three integers.

3.1.7 Smartblocks by Allwright et al.

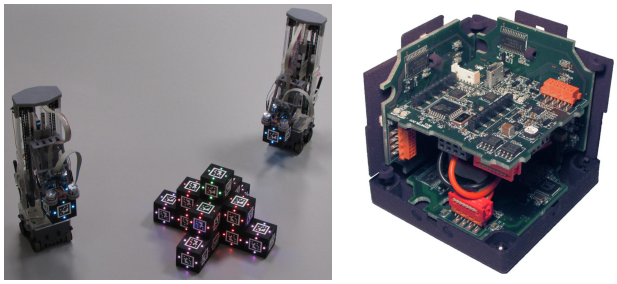


Figure 3.8: The Smartblocks constructing a 3D pyramidal structure (left); The inside of a Smartblock (right).

The Smartblocks - also called stigmergic blocks - are the semi-active blocks used as building material for an autonomous construction system (ACS) implemented by Allwright et al. [1][5] The control strategy for this multi-robot ACS is decentralized and is based on the work of Theraulaz and Bonabeau (see Section 2.1). The ACS in action is shown in Figure 3.8 (left).

The Smartblocks were designed to meet the requirements for this ACS. They contain freely-rotating spherical magnets in their corners. These spherical magnets allow simplifying actuation by aligning with the end-effector of an autonomous robot and by ensuring self-alignment, to a certain extent. They also strengthen the structures made from the blocks. The Smartblocks are markable by a robot using an NFC interface. The markings are displayed using LEDs in four different colours. These LED markings, along with the structural arrangement of the Smartblocks, form patterns in an environment that are detected by nearby robots. These patterns are used to trigger construction actions. A simplification of the sensing requirements for the moving robots is partially achieved by attaching localizable *April* tags to the Smartblocks, which enables an autonomous robot to accurately locate a block in an environment.

Additionally, the NFC can also be used for block-to-block communication. This allows experimenting with other algorithms where the blocks have a global knowledge of the structure and change their markings according to it. Block-to-block NFC communication is also envisioned to be used for remote monitoring of the structure it has to be enhanced first, which is the goal of the practical work of this master thesis.

3.2 Comparison

Tables 3.1 and 3.2 present a brief comparison of the discussed implementations over characteristics that were chosen because they have impact on the usability of the system for swarm construction. In the following, we detail five of those characteristics by explaining why they matter and how the different previously-presented blocks feature it or not.

Name	**Ref.**	**Goal**	**Mode of Assembly**	**Attachment**	**Power**	**Block-to-block Communication**
-none-	3.1.1	Render into virtual structures	Manual	DC connector and plugs	External, via outer pin of DC connector	Inner pin of DC connector
Boda Blocks	3.1.2	Educational toy	Manual	With a bar	-unspecified-	-unspecified-
Blinky Blocks	3.1.3	Affordable programmable matter supporting distributed language *MELD*	Manual	lego-like (vertical) + magnets (horizontal)	External, via electrical spring pin	serial link via spring pin
Robot Pebbles	3.1.4	Programmable matter through self-disassembly	Vibrating table & Magnetic attraction	EP magnet	External, via EP magnet + capacitor for high peek needs	EP magnet
M-blocks	3.1.5	Active building material	Self-assembly	magnets	Internal (4 batteries)	-none-
-none-	3.1.6	2D semi-active building material	Multi-robot	-none-	Internal	IR led blinking and photosensor
Smartblocks	3.1.7	Semi-active building material	Multi-robot	magnets	Internal	NFC

Table 3.1: Comparison of units used to construct intelligent structures.

Name	Ref.	Spec.	Sensors	Additional features	Shape	Size (cm)	Source
-none-	3.1.1	PIC16C77 microcontroller	-none-	-none-	Rectangular with pins on top	10 x 5 x 2.5	[10]
Boda Blocks	3.1.2	-unspecified-	A switch	Luminescent (in blue or green)	Cubic	-unspecified-	[11]
Blinky Blocks	3.1.3	Microcontroller with 80kB RAM	Microphone, orientation, sudden impulse (shaking or tapping)	LEDs, speaker	Cubic with pins on top	4 x 4 x 5	[12]
Robot Pebbles	3.1.4	ATMega328p, 2kB RAM	-none-	-none-	Cubic	1 x 1 x 1	[13][14]
M-blocks	3.1.5	nRF51422 microprocessor	-none-	Pivoting motion, and connection to remote computer with Xbee (v1), Bluetooth (v2), Infrared (future)	Cubic	5 x 5 x 5	[16][15]
-none-	3.1.6	-unspecified-	-none-	LED on top	Octogonal	-unspecified-	[3]
Smartblocks	3.1.7	ATMega328p, 2kB RAM	Accelerometer	LEDs, April Tags, connection to remote computer with Xbee	Cubic	5.5 x 5.5 x 5.5	[1][5]

Table 3.2: Comparison of units used to construct intelligent structures (continued from Table 3.1).

3.2.1 Attachment

Attachment is important to consider when we want the blocks to be manipulated by robots. Indeed, modules needing to be interlocked into others to form the structure require a lot more precision from the robots manipulating them. This is the case of the three first implementations (3.1.1, 3.1.2 and 3.1.3). This is coherent with the fact that all of those three blocks were designed to be manipulated by humans. In contrast, the blocks of Sugawara and Doi (3.1.6) provide no attachment method. The three others ensure attachment by the means of magnets. The Blinky Blocks also use magnets, but for horizontal alignment only. This system allows a margin of precision error by the manipulating robots, because blocks self-align when brought close enough from each other.

The Robots Pebbles (3.1.4), the blocks that use electro-permanent magnets for latching, can attract other modules (when aligned and when faces are parallel) from a distance of 2.48mm. When both blocks energize their magnets, the separation distance is increased up to 4.31mm. For the M-blocks (3.1.5), the ones capable of pivoting motion, attachment is ensured by twenty-four cylindrical magnets placed in the edges. While being a strong enough structural link, those magnets were not sufficient for self-alignment. Thus they are also provided with eight disc-magnets for this purpose, one at the centre of each face. At a distance of 5mm, the force provided by those magnets is roughly 2N. Finally, for the Smartblocks (3.1.7), the distance of attraction is 12mm.

Apart from that, the attachment method also influences the overall robustness of the structure and the possibility to form cantilevered arrangements.

3.2.2 Power Supply

Power supply, and in particular whether it is provided externally or embedded in the blocks, principally determine the infrastructures you need to have at disposal in the vicinity of the structure.

For external power, we never directly connect all the blocks in the structure. Instead, only one or few blocks are directly connected to the power supply and all other blocks need a way to connect electrically to them through the structure. With this scheme, the blocks cannot operate when disconnected from the structure: this limits the use of the computing abilities of the blocks to the aggregate state. This is the case for three of the seven hardware implementations presented here. Both the blocks by Anderson et al. (3.1.1) and the Blinky Blocks (3.1.3) are linked to their neighbours with electrical links. Their top/down connections were designed in order to respect polarity no matter the rotation of the upper/lower block. The blocks by Anderson et al. only have such connection vertically. Hence it restricts the kind of structures that can be made, as they need to be placed in an interleaving manner to achieve full connectivity. The Blinky Blocks have additional connections on their side through small rounded spring pins. The Robots Pebbles (3.1.4) use their EP magnets to pass power wirelessly from one block to another via Ohmic conduction. The choice of a wireless connection has the advantage to allow completely sealing the blocks, which could come handy for underwater use.

It is to note that the use of external power supply is also limiting from the fact that each block adds a small resistance. However, the limit seems to be quite big. For example, with the robots Pebbles, a 20V source could power a one-direction chain of 3266 modules. Moreover, the possibility to use multiple drains is probably feasible, while not being currently implemented.

On the contrary, the Smartblocks (3.1.7), the blocks by Sugawara et al. (3.1.6) and the M-blocks (3.1.5) with provided internal power storage. For the M-blocks, this choice was driven by the fact they are made to be capable of motion, during which they can be disconnected from the structure. The mechanism for providing movement developed here makes them also capable to move when alone in an open field, so a battery is needed for this case. While the advantage is the portability of the system, it induces a big drawback: battery life is limited in time. For example, the first version of M-blocks is limited to 20-100 motions. So for the case of autonomous construction in remote environments, both those two systems are limited, in fine, to the construction of static structures. In the future, when we will want to use such systems outside the laboratories, one would need to add a way for them to recharge autonomously - probably using renewable energy.

3.2.3 Shape

What is important about the shape, is whether it has rotational symmetry or not. The cubic-shaped blocks have this property. The blocks of Sugawara et al. (3.1.6) are to be taken apart: they do have rotational symmetry but only in 2D. In our list, only the blocks by Anderson et al. (3.1.1) and the Blinky blocks (3.1.3) don't have full rotational symmetry. They both have connectors and plugs on the top and down face so they do not support top-down inversion. But as said before, thanks to a good arrangement of the electrical connections, they support rotation along the vertical axis. The blocks by Anderson et al. have a rectangular shape so there is no symmetry, but it is because they don't have face connections so they must be placed overlapping several blocks in order to achieve full connectivity through the structure. The robot pebbles are neither rotational symmetric since they only have four EP magnets, as the current implementation is meant to be used only for 2D structures. However, their cubic shape was chosen for that purpose and it is planned to provide them in the future with six EP magnets.

With non-symmetric blocks, a pivoting manipulation would be needed as soon as a block falls and rolls, or if we want the blocks to be dropped randomly at the beginning of operations. Cubic blocks showing full rotational symmetry are more easy to handle, as they do not require the entity that manipulates them to be capable of rotating manipulation. Again, this feature is not relevant when the blocks are meant to be manipulated by humans. But when we want them to be moved by robots, it is a huge gain in simplicity.

3.2.4 Block-to-block communication

The bandwidth of block-to-block communication will determine the speed of message passing through the structure. We can divide block-to-block communication into two categories: wired or wireless.

As discussed for power supply (3.2.2), a wired connection requires the placement of the block to be exact, thus require more precision for the manipulating robots. The blocks by Anderson et al. (3.1.1) and the Blinky blocks (3.1.3) use wired connections. However, Blinky blocks horizontal connectors were designed as curved spring pins, meaning the connection is achieved by simple face contact - contact which is facilitated by the use of magnets ensuring self-alignment. The Boda Blocks (3.1.2) seem to use wired connection too, as they need to be connected with a bar, but no detail was given about that.

Four different technologies were used for wireless communication, all supporting the blocks to be spaced out to a certain extent. The blocks of Sugawara et al. (3.1.6) use an IR led and light sensors, where integer values can be passed by blinking the infrared LED. This is probably the most efficient connection in term of cost, as IR LEDs are widely used and manufactured nowadays. The Robots Pebbles (3.1.4) use their EP magnet for communication. They achieve 9600 bps communications by sending and sensing 1-microsecond magnetic pulses. The pulses must use the same polarity as the pulses used to latch the blocks to avoid decreasing the latching strength during intensive communication. This system would allow listening to message on multiple faces concurrently, but as the four EP-magnets share a common half-bridge, the modules are unable to distinguish between the faces. Therefore, they allowed a module to listen to only one face at a time. The Smartblocks by Allwright et al. (3.1.7) uses NFC peer-to-peer communication. At the time of writing, one communication takes roughly 250 milliseconds. Finally, the second version of the M-blocks (3.1.5) provides block-to-block communication with Bluetooth. They also already implement the *ANT* routing protocol.

3.2.5 Other abilities

Other abilities - meaning features that do not only serve for block-to-block communication - allow interaction with the environment, with a user, with a remote computer or with robots. Of course, the choice of additional features depends on the usage the blocks are intended for. When we want the blocks to be used as building material for swarm construction systems, it defines the means for stigmergy (2.1). One of the purposes of the use of semi-active building materials is to simplify the sensing requirements of the robots. Thus, in that case, the simpler to implement is the better.

Those capabilities fall into three categories: feedback mechanisms, sensors or two-way communication channel, for the pieces of information to go respectively out, in, and in/out. The most commonly observed feedback mechanism is through lights. Both the Boda blocks (3.1.2) and the Blinky Blocks (3.1.3) are translucid and have a LED inside them. On the other hand, the Smartblocks by Allwright et al. (3.1.7) have four RGB LEDs on each face, allowing them to give a per-face feedback. The blocks of Sugawara et al. (3.1.6) have

both: a per-face feedback with IR LEDs and a top led to give another feedback. Since the purpose of the stigmergic blocks is to guide the constructing robot in the deposit of other blocks in their neighbourhood, it is important to be capable to show distinct signals per face. Otherwise, this guidance might require the robot to process several more information (i.e. the colour of the other blocks) or might be limited to a certain class of structures. However, the stigmergic blocks are obviously still suitable for testing algorithms that don't need per-face feedback, as demonstrated in [5]. Additionally, the Blinky Blocks are provided with a speaker for sound.

The sensors make the blocks able to receive information from their environment. As explained in 2.2.3, adding sensors can lead to interesting uses of intelligent structures since they can sense a hazard and react to it. However, here, sensors are mostly used for communication. The Boda Blocks only have a simple switch that allows users to change their state. The Blinky blocks are the most furnished: they have a microphone and an accelerometer. The Smartblocks (3.1.7) also present an accelerometer. The accelerometer can be used to sense sudden impulse, like shaking or tapping, and to sense when they are being moved. It is also used to distinguish up and down. The Blinky Blocks use it to prevent incorrect orientation, as they do not support top-down inversion.

For what concerns interactions with a remote computer, it must be made with a two-way communication channel like wifi, Bluetooth or UART. Similarly to power distribution, to avoid loads of cables or to avoid adding a wireless chip in every block because communication with a remote computer is not the main feature, a common scheme is to connect only one of the block. Then, there needs to be a message passing protocol implemented over the block-to-block communication, and the structure is not homogeneous anymore. This is the case for the blocks by Anderson et al. (3.1.1), for the Blinky Blocks and, in the future, for the Smartblocks. The Robot Pebbles (3.1.4) also use this scheme but only for a start signal, that will power up the whole structure thus start the program. Also, the Robot Pebbles are the only ones that use the so induced heterogeneity. Indeed, at the start of the system, the blocks learn their position in a coordinate system and the block receiving the start signal is used as the origin of this system. The M-blocks do not use message passing since they are all provided with a wireless communication chip. Indeed, for them, connection with a remote computer is a main feature since, at present, their movement must be ordered by a user. The first 2D version used an Xbee Wifi module, the second version used Bluetooth, and in the future they plan to use an infrared connection.

Finally, for the Smartblocks, while NFC is the means for block-to-block communication, it can also be used for robot-to-block communication. It allows, for example, the manipulating robots to change the colour of the block before placing it in the structure. It would be possible to achieve that with other block-to-block communication means, although it would require more precision to connect pins than to draw two NFC chip close enough to each other. However, NFC is not used for block-to-robot communication because it would require the robots to get close to each block before getting information, thus eventually causing congestion. With light signals and a camera, robots can get information from several blocks at a time from a fair distance.

3.3 Conclusion

We reviewed seven existing implementations of building blocks. They were all designed with different goals in mind, but as they are all capable of communicating with each other, they have the potential to be used as semi-active blocks for autonomous construction with swarm robotics. However, in many cases, the high manipulation precision and sensing requirement needed by the moving robots would prevent feasible practical implementation.

First, swarm construction benefit from the use of semi-active blocks as building material only if they feature a feedback mechanism to guide the manipulating robots. Additionally, this feedback must be practical enough to allow simplify sensing requirement of the manipulating robots. The blocks by Anderson et al. (3.1.1), the Robots Pebbles (3.1.4) and the M-blocks (3.1.5) do not provide any feedback mechanism. For them, block-to-block communication could be used but at the extent of a great loss of efficiency, as the robot needs to place himself next to the block to sense the feedback, thus causing congestion and many additional displacements. The most common feedback mechanism, with light, is what we find in the four remaining blocks. However, the Blinky Blocks (3.1.3) and the Boda Blocks (3.1.2) feature only one LED (the latter with only two color), whose color can be seen on every face, while the Smartblocks by Allwright et al. (3.1.7) and the blocks of Sugawara et al. (3.1.6) provides a per-face lightning by having different LEDs on each of their faces. In autonomous construction with swarm robotic, having per-face signal is advantageous because it is a requirement for several algorithms, notably the ones presented in [3] and [7]. For those two blocks, the LEDs are used differently. The blocks of Sugawara et al. sends integers via the blinking rate of their infrared led, and the Smartblocks use the variation of RGB colours to send information about their state. This latter allows a great variety of signals, while it has the drawback that it requires computer vision abilities. Finally, it is to note that the Blinky Blocks also feature sound feedback with speakers, but this is also one signal for the whole block.

Secondly, in order to be manipulated by robots, the blocks must provide features that facilitate placement. The first three, the blocks by Anderson et al. (3.1.1), the Boda blocks (3.1.2) and the Blinky blocks (3.1.3) needs to be pinned one into another to pass messages, power, and to form the structure. It provides a great structural solidity, but it induces higher precision requirements from the manipulating robots. This drawback comes from the fact that they all were designed to be manipulated by humans, thus resistance to a certain lack of precision was not a requirement. The blocks of Sugawara et al. (3.1.6) does not provide any structural link, nor to ensure self-alignment nor to ensure structural solidity. Finally, the three remaining blocks, the robots Pebbles (3.1.4), the M-blocks (3.1.5) and the Smartblocks by Allwright et al. (3.1.7) all enable self-alignment with magnets.

Another way to simplify the capabilities of the manipulating robot is by using rotational symmetric blocks. Indeed, a block that must respect a certain orientation may require the manipulating robot to be able to rotate it. On the other hand, a rotational symmetric block can be picked up as it is placed, can be dropped and picked up again, etc. However, we talk about rotational symmetry here while talking about cubic blocks, so it would still

eventually need a partial rotation capability to place the blocks correctly. However, it will be straightforward and cheap if the block provides self-alignment by magnets, as this capability then only requires the arm of the robot to contain magnets too. In our list, the Boda blocks (3.1.2) and the Smartblocks by Allwright et al. (3.1.7) are cubic thus respect this requirement. However, the robots Pebbles (3.1.4) and the blocks of Sugawara et al. (3.1.6) were designed for 2D structures so further developments are to follow. The robots Pebbles already have a cubic shape but implement block-to-block communication only on four faces.

Finally, the blocks should feature efficient block-to-block communication. The bandwidth of block-to-block communication will determine the speed of message passing through the structure. The problem with a slow communication is that during the time where the message is not yet received by a particular block, it will cause it to display obsolete information, thus causing errors in the construction process. While the error can surely be handled, it will slow down the construction and we want to avoid that. Instantaneous communication is not achievable, but we can reduce the delay by using fast communications. Unfortunately, not many papers give the bandwidth value achieved with their implementation. We can only say that the Robots Pebbles (3.1.4) are capable of 9600 bps communications and that the Smartblocks take roughly 250 milliseconds to send a message via NFC. Additionally, the practical work made as part of this master thesis aims to improve the speed of a message passing inside a structure of Smartblocks by implementing multitasking. It is to note that the Robot Pebbles uses the same microcontroller as the Smartblock, so the multitasking software could eventually be reused for them.

Name	Section	Feedback Mechanism	Placement Facility	Rotational Symmetry
Blocks by Anderson et al.	3.1.1	none	difficult	no
Boda Blocks	3.1.2	visual	difficult	yes
Blinky Blocks	3.1.3	visual	difficult	no
Robots Pebbles	3.1.4	none	easy	no (2D)
Blocks of Sugawara et al.	3.1.6	visual, per-face	medium	no (2D)
Smartblocks	3.1.7	visual, per-face	easy	yes

Table 3.3: Shorter comparison of units used to construct intelligent structures, summarizing the conclusion.

The table 3.3 summarizes what was said in the previous paragraphs by classifying all the building blocks according to the required qualities. From this, we may infer that the Smartblocks are the best suited. The blocks of Sugawara and Doi could suit too, but not for our case as they are designed for 2D structures. The others either provide no visual feedbacks, either are difficult to place, either are not rotationally symmetrical. Of course, those weaknesses could be overcome, but it would induce an increase of complexity and thus of cost.

Part 4

Development of the Software

The following details the multitasking software that was developed as practical work for this master thesis. As explained earlier, by enabling simultaneous communication, we want to increase bandwidth and thus allowing an efficient routing of messages to be implemented. The final software does not reach this goal as it suffer from critical issues, as explained in Section 5.1. The code is available at `github.com/romische/Smartblocks-os`.

We divided this section into three parts. The first, Section 4.1, explains the multitasking system. The second, Section 4.2, details each controller driving a Smartblock's feature. And the third, Section 4.3, shows how the global behaviour of the system can be defined in a main file.

4.1 Multitasking System

4.1.1 Preemptive vs. Cooperative Multitasking

There are two ways to implement multitasking, depending on who has the responsibility to initiate the task switching. When it is initiated by the tasks themselves, it is called *cooperative multitasking*. The drawback is that it is complex to handle for the developer, as it means adding the instruction for task switching at the right place. With *preemptive multitasking*, the task switch is triggered by the system at regular interval. The drawbacks are that it consumes an interrupt, a resource that can be sparse on microcontrollers, and that we need to implement a way to prevent critical resources to be used by several tasks in the same time. We chose to implement this latter here.

Preemptive multitasking implies that a task can be interrupted at any time, even in the middle of an operation. For some parts, such interruption is unwanted. Those can be protected by placing them inside "`cli-sei`" instructions. `cli()` disables the interrupt, and `sei()` re-enables them. However, if used inside another "`cli-sei`" part, it will cause the interrupts to be enabled too early. To avoid that, it is better to save the content of the status register (SREG), then disable the interrupts, then reset the status register to its saved value. We tried to avoid disabling the interrupt as much as possible, as it results in uncertainty in timing as explained in 5.1.3.

4.1.2 Inspiration

Our implementation is greatly inspired by the library *avr-os*. This library was written in 2012 by Chris Moos and can be found in his *Github* repository at `github.com/chrismoos/avr-os`.

avr-os uses the language C to implement pre-emptive multitasking. It also provides mutexes and spinlocks, that can be used by a task to prevent others to access a variable when concurrent access is unwanted. It is compatible with several microcontroller boards, including the Redboard from Sparkfun. The Redboard has the same processor as our Smartblocks: the ATMega328p from Atmel.

4.1.3 General Idea

The general idea is as follows: The user can create tasks just by defining a method in the main file or in another class. It passes the address of these methods to the `System` instance using the function `schedule_task`. If the system has not reached the maximum number of tasks it can handle, it adds it to its list of tasks and allocates a portion of the RAM to it. In order to start, a task just needs to have this address and the address of the eventual arguments registered onto the stack. This mimic the way functions are called in a normal execution flow.

Then the user calls the `run` method. This will cause the first task to start and will initiate an interrupt that occurs every 10ms. If there is more than one task running, this interrupt will execute a context switch as explained below and so pass the execution to another task. Moreover, a task can also use the functions `yield` or `sleep` to directly pass the execution to the next task without waiting for the interrupt.

4.1.4 Methods Provided by the System Object

The whole multitasking system is contained in the C++ class `System`, composed of the two files *system.h* and *system.cpp*. This object implements the singleton pattern, meaning there can only exist one instance in the whole program. The constructor is not accessible and we can obtain this instance only with the function `instance()`.

Apart from this function, the object `System` provides five public methods :

- `run` starts the multitasking environment, by starting the first task and enabling the interrupt. It must be called only once.
- `schedule_task` serves for adding a task. It can be used at any time. However since the number of tasks is limited, it is possible that the task gets refused. If so, the function returns the error code -1. `schedule_task` takes two void pointers as arguments: the

address of the function corresponding to the task and the address of the arguments for this function. The function itself must respect some constraints that are enumerated in Section 4.3

The three others are to use from inside a task. They can not be used outside the multitasking environment as it will cause errors otherwise.

- `yield` passes the execution to the next task.
- `sleep` passes the execution time to the next task and prevent the current task to execute during a time equal to the value passed in argument.
- `exit_task` ends the current task.

4.1.5 Parametrization of the System

We can parametrize the system using three macros at the top of the file *system.h* as illustrated in listing 4.1.

```
#define TASK_STACK_SIZE 256
#define MAX_TASKS 3
#define TICK_INTERVAL 10
```

Listing 4.1: The macros definitions used to parametrize the system

`TASK_STACK_SIZE` defines the size, in bytes, of the portion of the RAM to reserve for one task. The same amount of RAM is let for use for the main loop and the code that occurs before the start of the multitasking environment. The bigger its value is, the less we risk that a task overwrites the RAM space of another task. And the smaller it is, the more tasks we can run. We chose to keep here the value used in *avr-os*. However, the user may want to change it depending on the type of tasks he defines and how much RAM they require. We advise to choose the smaller value that doesn't provoke overwrites errors.

`MAX_TASKS` define how many tasks can be run concurrently by the system. It is the role of the user to set this variable to the right value, according to the value chosen for `TASK_STACK_SIZE`, and to the portion of RAM space that can be used for the stack as explained in the following.

The *ATMega328p* provides 2048 bytes of RAM, whose addresses goes from 256 to 2303. The RAM usage is illustrated in Figure 4.1. A consequent amount of it is consumed by the segments *text*, *data* and *bss*. Those segments respectively store the constant, initialized and uninitialized data. Then the heap is used for dynamical allocation of memory, notably using the `malloc` method. It grows in increasing order of memory addresses. Finally, the stack is used to store information relative to the execution of the software. It includes, for example, local variables, arguments and return addresses.

In our case, we want to divide the stack into `MAX_TASKS` blocks of `TASK_STACK_SIZE` bytes to provide each task with its own private stack. The RAM space we can use for that purpose is the space left between the top of the heap and the top of the stack. There is no

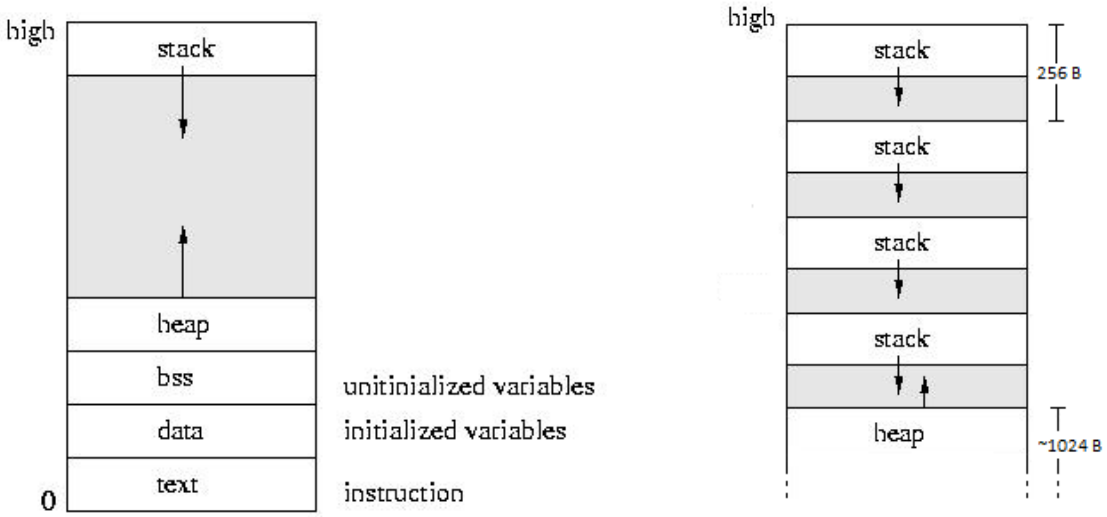


Figure 4.1: (left) The RAM usage in a single-tasking system; (right) The RAM uasge in our implementation of a multitasking system.

global variable storing the address of the top of the heap, but we can get an estimation of it by using by looking at the address of the pointer that is returned by the `malloc` function. Or, alternatively, the top of the *bss* section is stored in the global variable `__bss_end`. For the stack, the stack pointer stores the top of the stack. It is to be fetched using some assembly code as shown in listing 4.2. Upon creation, the `System` object stores the stack pointer in a variable called `stackTop`. It then reserves `TASK_STACK_SIZE` bytes for the main loop and create a variable called `tasksStackTop`. This latter indicates the place from which the task's stack can begin. Therefore, by looking at the difference between `tasksStackTop` and the estimated top of the heap, the user can infer how much bytes are free to use for the tasks. Finally, `MAX_TASKS` must be chosen such that `MAX_TASKS`*`TASK_STACK_SIZE` can fit in this free space. For us, it led to a value of 3 tasks.

```
asm volatile(   "in r0, __SP_L__" \
                "sts stackTopCheck, r0" \
                "in r0, __SP_H__" \
                "sts stackTopCheck+1, r0"  );
```

Listing 4.2: The inline assembly code used to fetch the stack pointer in a variable called `stackTopCheck`.

We make two key assumptions here. First, we assume that the heap will not grow. It will be the case if the user avoids using the `malloc` method. The code base doesn't use it. Secondly, we assume that the stack will not grow too much between the time the `System` object is created and the time the multitasking system is started by the method `run`. For this, the user must keep the code before the `run` method as simple as possible. This latter issue is not present in the *avr-os* software as it tackles the allocation of RAM space for the tasks a bit differently. The `tasksStackTop` variable is defined when calling the `run` method instead of upon creation of the `System` object. We chose to change this as it allowed us to reduce the space used by a `task_definition` structure (see Section 4.1.6).

Alternatively, to choose the correct value for the `MAX_TASKS` parameter, the user can just test and observe. If the number is too big, the stack of the tasks will overwrite the heap and eventually the *bss* section. This result in undefined behaviour. Common clues are data corruption and the software restarting indefinitely.

Finally, `TICK_INTERVAL` set the time interval between two interrupts. A small value increases the frequency of the interrupt, and a bigger one decreases it. There is no critical functional problem caused by the choice of a big value. However, it will limit the efficiency of the concurrency and increase the time uncertainty (see Section 4.1.8 and 5.1.3). On the other hand, a value too small could corrupt the system. Indeed, the interrupts need some time to execute. With a tick interval that is extremely short, a new interrupt will occur as soon as the previous one has terminated, thus letting no more - or few - execution time for the rest of the program. We chose to keep here the value used in *avr-os* as it was found to work sufficiently well.

4.1.6 Definition of a task

Tasks are defined as structures, named `task_definition`. This definition is to be found at the beginning of the *system.h* file. The structure only containing three elements :

- `sp` A pointer to the stack. As explained below in section 4.1.7, when interrupting a task, this variable holds the address of the top of the stack where the context has been saved. When restarting the task again, this address is used to fetch the context from the stack.
- `running` A boolean indicating if the task is running or not. If it is set to false, it indicates that the task has terminated and thus that it can be removed and that its RAM space can be used for a new task. It is set to false by the function `exit_task`.
- `sleep_time` An integer indicating how many milliseconds the tasks need to sleep. This value is set by the method `sleep` and is decremented in the interrupt as explained in Section 4.1.8.

It is less than in the original software, as we made some simplifications. The first simplification was to not let the possibility to delay the start of the task. When one uses `schedule_task`, the task is added to the list of running tasks rightaway.

The second simplification is to directly allocate and prepare RAM space for the task. In *avr-os*, we had to store the address of the method defining the task and the address of its arguments in order to later register those to the RAM. This has the advantage of letting more flexibility for the RAM usage before starting the multitasking environment. Indeed, in our case, the code before the call to the `run` method is allocated a certain amount of bytes (the same as what is allocated for a task) and no more. If it uses too much RAM, it is possible that it overwrites the addresses written on the stack for starting the first task. Then this task will not be able to start. However, this problem can occur later in the execution. It is explained in the section 5.1.1.

4.1.7 The Context Switch

The context of a task consists of partial results and computer register contents (see Figure 4.2). When switching from one task to another, we need to save the context of the current task onto its stack and retrieve the context of the next task from its stack to the registers. In practice, it is accomplished with two macros defined in the file *context.h*. These macros use inline assembly code and in particular the instructions `pop` and `push`.

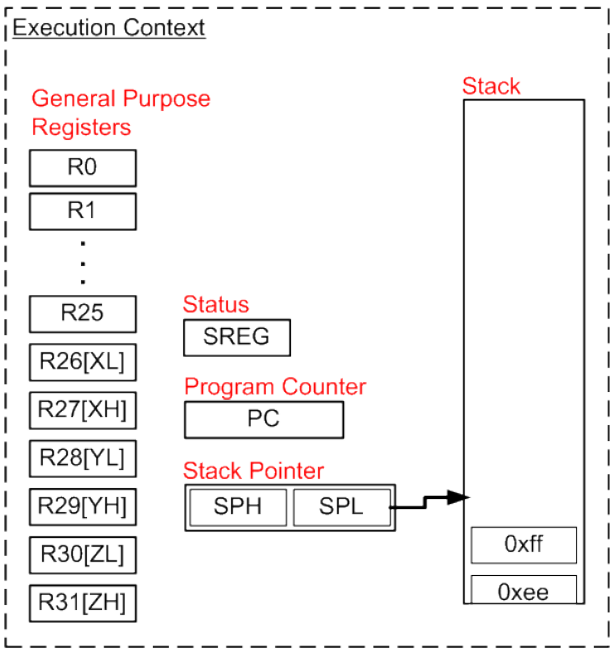


Figure 4.2: The context of a task in an AVR microcontroller. From `www.freertos.org`

The macro `SAVE_CONTEXT` pushes the 32 registers and the status register onto the stack. Then it retrieves the address of the top of the stack from the global variables `SP_L` and `SP_H`, and saves it to the stack pointer passed in argument. This stack pointer is the variable `sp` from the current task.

The macro `RESTORE_CONTEXT` does the opposite. It sets the global variables `SP_L` and `SP_H` to the stack pointer of the task we want to run afterwards. A proper stack pointer is needed for the `pop` instruction to fetch value at the correct location. Then it pops the 32 registers and the status register that were previously pushed onto the stack in reverse order.

The program counter is the last element composing a context. It points to the next instruction to execute. However, it is not accessible via a global variable as the stack pointer. But we do not have to take care of it because it is automatically stored and restored from the stack when starting and terminating a function or an interrupt routine. Thus we do not have to handle it ourselves. As long as the stack pointer has been changed correctly, the program counter will be retrieved from the correct memory location.

Between those two instructions, we have to choose which task is the next to execute. Because we did not implement any form of priority for the task, we use here a simple

"Round-Robin" scheduling algorithm. That is, all tasks are assigned the same amount of execution time in circular order. Of course, the tasks that have terminated (boolean `running` set to false) and the tasks that are paused (integer `sleep_time` greater than zero) are ignored. If no task respecting those conditions is found, we pass execution to the main loop, that is a loop doing nothing. This main loop was started at the beginning of the multitasking environment and its context is pointed out by the global variable `stackTop`.

All tasks are referred to via a simple integer, which corresponds to the index they occupy in the array containing the `task_definition` structures. The main loop corresponds to index -1. We keep track of the current task only via an unprotected global variable. If this variable gets modified other than here in the context switch, it will lead to errors. That is one typical weakness for which we should implement the whole `System` object with more respect to the C++ design.

The context switch can be triggered at two different occasions. First by the interrupt occurring every 10 milliseconds, and secondly, the context switch can be initiated by a task with the `yield` or `sleep` function. Because the interrupt has an additional responsibility, as explained in the next section (4.1.8), we had to create two separate functions for those two cases.

To link our interrupt to a behaviour, we use the ISR (Interrupt Service Routine) macro defined in *avr/interrupt.c*. We use here a naked interrupt. This prevents the service routine from saving and restoring the context that has been interrupted, respectively at the beginning and the end of the routine. Indeed, we want to take control of this work, in order to change the context to restore. If the instruction was not naked, the `SAVE_CONTEXT` instruction would occur only after the ISR has changed the context, thus we would save the context of the interrupt instead of the context of the task being interrupted.

4.1.8 Management of the Time for the Sleep Function

A task can use the function `sleep(t)` to pause itself during t milliseconds. When called, this function will just set the attribute `sleep_time` of the current task to the chosen delay and pass the execution to the next task by doing a context switch.

Then, this value is reduced by 10 - or by whatever is the value of the macro `TICK_INTERVAL` - each time the interrupt routine occurs. A task is ready to restart its operation when its `sleep_time` parameter gets lower than zero. This decease of the value of `sleep_time` is only done when the context switch is triggered by the interrupt.

As explained in section 5.1.7, this way of working has a precision of 10ms and not less. For example, using `sleep(1)` may provoke the task to sleep for up to 10ms. A probably more precise way was implemented in the *avr-os* software. However, we chose to change it because it was consuming one task, which is a sparse resource in our implementation. Another drawback of our way of working is that the interrupt takes more time to execute. Section 5.1.7 proposes more solutions to these problems.

4.1.9 The Locking Mechanism: class Resource

Although the system executes tasks concurrently, some resources cannot be used concurrently. For example, if we send two words through the serial communication at the same time, they will be outputted in an interleaved way. In general, concurrent access causes more severe errors than that. For such resources, concurrent access must be prevented by a locking mechanism.

In our system, it is the case for all the controllers presented in Section 4.2. Therefore, they all inherit from a class called `Resource` implemented in the file *resource.h*. This class provide two methods : `lock` and `unlock`. When a resource is locked by a task, other tasks cannot access it. Instead, they wait indefinitely for the resource to be unlocked. It is the role of the programmers to ensure that the resource gets unlocked at some point. To do so, they first must avoid using an infinite loop inside a lock-unlock sequence. Secondly, they must avoid the *deadlock* problem. This problem arises when one task locks a resource A, then needs to lock a resource B, but cannot get it because resource B is locked by another task and this latter is blocked because it needs the resource A to continue its execution. Therefore, the two tasks will forever wait for each another. To avoid this problem, we advise using one resource at a time (unlocking a resource before locking a new one). If not possible, it is better to lock resources always in the same order.

The lock is simply a boolean set to true or false. The modification of this value is to be made after disabling the interrupts, to avoid a task to be interrupted in the middle of this operation. Indeed, this could result in two tasks both getting the authorization to use the desired resource.

In order for the spinlocks to function correctly, the objects inheriting from the class `Resource` must be implemented as a singleton. The singleton pattern ensures that one and only one instance of the object will be created in the whole system. Indeed, if we allow several instances to exist, there will also exist several values for the lock, and it will result in unwanted concurrent access.

4.1.10 The Timer

The timer is an object counting the time in milliseconds and it provides only one public function: `GetMilliseconds`. Internally, it works with an interrupt that occurs every millisecond and triggers the incrementation of a counter. It is also implemented as a singleton because the count of milliseconds must be unique. Since it is a read-only object, there was no need to apply the locking mechanism to it. Therefore, it does not inherit from the class `Resource`.

4.2 Integration of the Controllers

We have five controllers, one for each critical resource of the Smartblock. The code governing them was taken from the previously existing software, with slight modifications to adapt them to our multitasking environment.

4.2.1 Hardware of the Smartblocks

The hardware of the Smartblocks is explained in details in the technical report [5]. We present here a basic overview that will help us understand the needs and workflow process for the software implementation.

The Smartblock is composed of one central circuit board and 6 face boards. The central board contain the ATMega328p microcontroller, an accelerometer, a USB port, a socket for an Xbee wireless module, a battery plug and six connectors to the six face boards.

The **USB port** is connected to the serial port of the microcontroller via a USB-to-serial IC converter to allow communication with the Smartblocks from a computer. In practice, on the computer, we use the *picocom*[1] software to emulate a simple terminal that sends data over the USB connection. The converter also can detect if a battery is plugged in and charge it if needed. Moreover, the serial link can be used for reprogramming because of the chosen bootloader (*Optiboot*). The optional **Xbee wireless module** provides an alternative way to communicate with the Smartblocks. It is connected to the controller via an emulated serial port.

Each face circuit board contains a **NFC** transceiver and a **LED** driver. The LED driver is used to set the brightness of the red, blue, and green channels of four multi-colour LEDs on a face circuit board. The NFC transceiver allows messages to be sent and received wirelessly to nearby robots or blocks.

The six connectors to the face circuit boards provide each of them with power, an interrupt line and a **I2C bus**. I2C respects a protocol to send *Read* or *Write* transactions to peripherical ICs. Since all faces circuit boards are identical, connecting them all directly to the same bus would bring address conflicts. Therefore, we segment the I2C link into six different buses, one for each face. However, the microcontroller is only connected to one at a time. We can select the bus by setting the global attribute `PORTC` to the correct value. We call these segmented buses *ports*. A seventh port exists, for the connection to ICs that are internal to the central board (i.e. an I2C controller and the accelerometer).

Finally, the table 4.1 match each device with the name of the chip used to implement it.

[1] `https://linux.die.net/man/8/picocom`

Microcontroller	ATMega328p
USB-to-serial converter	FT231X
Power management	BQ24075
Wireless module	Xbee
NFC transceiver	PN532
LED driver	PCA963X
Port controller	PCA9554

Table 4.1: Name of the chips corresponding to the devices present in the Smartblocks.

4.2.2 Overview of the Controllers

Following the explanation from the previous section, we would need eight controllers. However, three of them were considered less essential and were not included yet in the software. They are shown in grey in Figure 4.3. The accelerometer is already used in the project, but we did not feel the need to implement it as a controller yet.

- **CTUARTController** would represent the emulated serial port for the Xbee wireless interface,
- **CADCController** would let the user retrieve the battery charge level via an analog-to-digital converter,
- **CAccelController** would control the accelerometer.

The five other are needed to run the basic functions of a Smartblocks. They are detailed in the following sections.

- **CLedController** for the faces Leds driver,
- **CNFCController** for the faces NFC tranceiver,
- **CHUARTController** for the communication link over usb,
- **CPortController** for the PORTC variable and the interrupt lines,
- **CTWController** for the I2C buses.

The Figure 4.3 shows the links between these classes. This figure does not represent a canonical class diagram because all the controllers are implemented as singleton or multiton. This means that the instance of an object is contained by the object itself. Therefore, the only link our diagram shows is the use of an object by another. To use an object that is a singleton, we only need to "include" its header file and use the method `GetInstance` or `instance` to retrieve the unique instance.

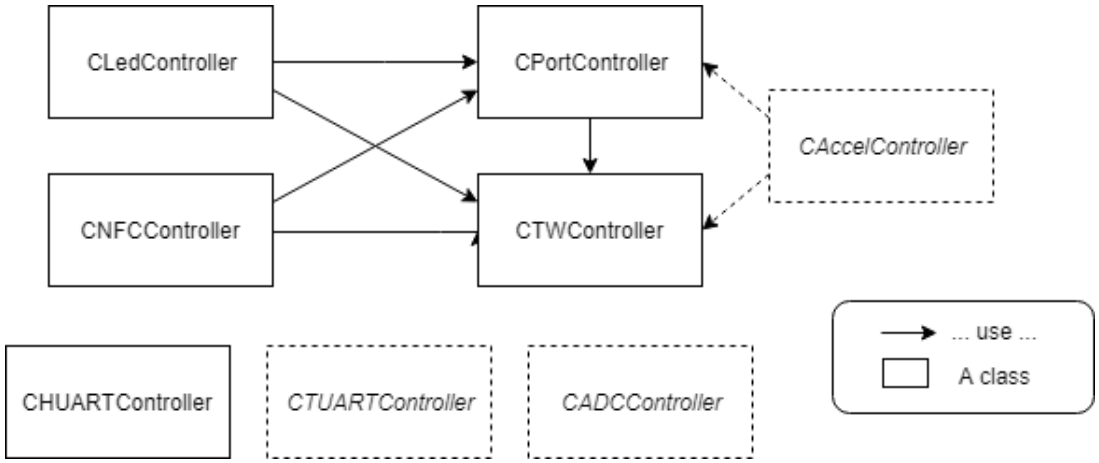


Figure 4.3: Simplified class diagram showing the controllers objects. Those not yet included in the software are shown in dashed lines.

4.2.3 The HUART Controller

The HUART Controller controls the serial link that allows the Smartblocks to communicate with a computer over a USB cable. It is implemented as a class named `CHUARTController` in the files *huart_controller.h* and *huart_controller.cpp*. As shown in Figure 4.3, the HUART controller uses neither the TW Controller, neither the Port Controller. It entails that apart from making it inherit from the `Resource` class (section 4.1.9), no additional modifications were required to the file provided by M. Allwright.

It uses two 64-bytes ring buffers to store respectively bytes that are received from the computer and bytes that need to be transmitted to the computer. The storing of data in the input buffer is driven by an interrupt, itself triggered when a user presses a key on the computer. The "unloading" of data from the transmit buffer to the computer similarly works with an interrupt. Consequently, the writing and reading operations are simplified to writing and reading from the two buffers.

The HUART object also stores a file object, accessible via the function `get_file`. It allows sending more than one byte at a time by using the standard method `fprintf`.

4.2.4 The TW Controller

As briefly stated before, the TW controller manages the I2C bus. It is implemented as a class called `CTWController` in the files *tw_controller.h* and *tw_controller.cpp*. It ensures the protocol of I2C communication is respected. Basically, every communication must begin with a start signal and terminates with a stop signal. The start signal is implemented by the `BeginTransmission` function that takes as argument the address of the device we want to communicate with. The stop signal is sent either by calling `EndTransmission(true)`, either by passing `true` to the `Read` function. To illustrate this, two examples of transactions are shown in listings 4.3 and 4.4 hereunder. The number of call to the functions `Write()` and `Read()` vary, but generally, the structure stays the same.

```
CTWController::GetInstance().BeginTransmission(device_address);
CTWController::GetInstance().Write(byte);
CTWController::GetInstance().Write(byte);
CTWController::GetInstance().EndTransmission(true);
```

Listing 4.3: A typical send operation with the CTWController.

```
CTWController::GetInstance().BeginTransmission(device_address);
CTWController::GetInstance().Write(byte);
CTWController::GetInstance().EndTransmission(false);
CTWController::GetInstance().Read(device_address, 1, true);
CTWController::GetInstance().Read()
```

Listing 4.4: A typical receive operation with the CTWController.

Those basic operations should not be interrupted by another use of the TWController, thus we must encapsulate them in a "`lock-unlock`" statement. Nonetheless, our testings seem to show that it can be interrupted by other operations that do not use the I2C bus, so there is no need to disable the interrupts for them. However, there is a slight possibility that the interrupt never occurred at a problematic moment and that it will be in the future.

For now, these instructions are utilized by other objects in the way that is shown in listings 4.3 and 4.4. But since there is indeed a general scheme of use emerging, it could be a better idea to encapsulate them in a higher level function. It would enhance the readability of the code and reduce a bit the number of lines. We did not do it here because the TW controller use it a bit differently and we wanted to keep homogeneity.

Finally, it is important to note that the TW controller uses two buffers to store incoming and outgoing messages. The filling/emptying of those buffers is triggered by an interrupt. This interrupt, among other responsibilities, is also in charge of changing the state of the TW controller stored in the global variable `unState` and of updating the error code stored in the global variable `unError` if an error occurred.

4.2.5 The Port Controller

The port controller handles both the interrupts coming from devices that are connected to the I2C and handles the `PORTC` variable that determines to which face the I2C bus is connected. It is implemented as a class called `CPortController`, in the files *port_controller.h* and *port_controller.cpp*.

In the following, a face to which the I2C bus can connect is called a *port*. There exists seven of them that the port controller defines in an enumeration: North, East, South, West, Top, Bottom and Nullport. The latter is the one to which the devices of the central board are connected, i.e. the accelerometer and the port controller itself. It must be selected by default at the beginning of the program to ensure no message is passed to a face that is actually not physically connected. Otherwise, it would cause the TW controller to stall.

To select a port, one must simply use the function `SelectPort` just before the use of a TW operation. As we must wait for at least one millisecond for the modification of the `PORTC` variable to be effective, this must be followed by a `sleep` instruction. To ensure that the port will not be modified during the execution of it, causing a part of the instructions being sent to another face, we must properly lock the port controller object and unlock it after the TW operation. For example, TW operations as shown in the previous section should be surrounded by lock instructions as shown in listing 4.5.

```
CPortController::instance().lock();
CPortController::instance().SelectPort(CPortController::EPort::
  NORTH);
System::instance().sleep(1);
CTWController::GetInstance().lock();
// TW instructions
CPortController::instance().unlock();
CTWController::GetInstance().unlock();
```

Listing 4.5: A TW operation surrounded by correct lock and unlock instructions.

The other role of the port controller is to handle the interrupts on the I2C bus. As said before, the I2C provides all the devices that are connected to it with an interrupt line. In practice, only the NFC transceiver uses it. As explained in Section 4.2.7, it is parametrized to send an interrupt when it wakes up from low power mode and this awakening occurs notably when it senses a RF field which indicates that another block is trying to communicate.

When an interrupt occurs, a one-bit flag is set in the register state of the port controller. It indicates from which face the interrupt is coming from. As retrieving this value require to use the TW controller, we avoid doing so from inside the interrupt routine. Instead, the interrupt routine only set a boolean indicating that an interrupt has occurred to true, then the user must use the method `SynchronizeInterrupts` to retrieve the value of the register state if needed. This value is then stored in a class variable by taking care to not erase older flags in the process.

The user can then retrieve the value of the status register with the function `GetInterrupts` . It must then use a loop to check the value of the flags one by one. A flag set to one at the position x indicates that an interrupt was sent by the NFC transceiver on the face whose enum value corresponds to the number x. As shown in listing 4.6, this leads to a pretty huge amount of code lines just to detect which face has sent an interrupt. A suggestion for the future would be to, in order to enhance clarity, "hide" this code into a member function of the port controller. This method would directly return the enum element corresponding to one of the faces from which an interrupt came from. And it is to note that the current implementation will always favour the faces with a low corresponding enum value.

```
CPortController::instance().lock();
CPortController::instance().SynchronizeInterrupts();
if(CPortController::instance().HasInterrupts()) {
   //fetch the interrupt code
   uint8_t unIRQs = CPortController::instance().GetInterrupts();
```

```
    CPortController::instance().ClearInterrupts(unIRQs);
    CPortController::instance().unlock();

    //for each face
    // (The list m_peConnectedPorts contains all the ports that
   are connected)
    for(CPortController::EPort eConnectedPort :
   m_peConnectedPorts) {
        if(eConnectedPort != CPortController::EPort::NULLPORT) {
            //on the face corresponding to the interrupt
            if((unIRQs >> static_cast<uint8_t>(eConnectedPort)) &
    0x01) {
                /*activate the NFC transceiver for reception*/
            }
        }
    }
}
else{
    //unlock even if the port controller has no interrupt
    CPortController::instance().unlock();
}
```

Listing 4.6: The code used to detect which face has sent an interrupt.

Finally, to know if a face is connected or not, the port controller provides the method `IsPortConnected`. In our case, it is used in the main file to construct the list `m_peConnectedPorts` containing all the ports that are connected.

4.2.6 The LED Controller

The LED controller is implemented as a class called `CLEDController` in the files *led_controller.h* and *led_controller.cpp*.

Every face contains four "packs" of LEDs, each containing three LEDs (red, green and blue). For each LED, we can control the mode and the brightness. Four modes are available : on, off, pulse-width modulation (*pwm*) or blinking mode. The *pwm* mode allows controlling the brightness of the light as with a dimmer. For the blinking mode, the period and the duty cycle (i.e. the percentage of the period during which the led is on) must be set beforehand with the method `SetBlinkRate`. These parameters can be set by sending commands to the PCA963X chip via the I2C bus. We need to correctly lock and unlock the port controller and the TW controller, as shown before in listing 4.5.

4.2.7 The NFC Controller

The NFC controller is implemented as a class called `CNFCController` in the files *nfc_controller.h* and *nfc_controller.cpp*.

The NFC communication is handled by a *PN532* chip. This chip supports, among others, the DEP protocol for the NFC exchange. A time-sequence diagram of a communi-

cation between two devices is shown in Figure 4.4. We define the two devices operating the communication as the *initiator*, the one that sends a message, and the *target*, the one that receives it.

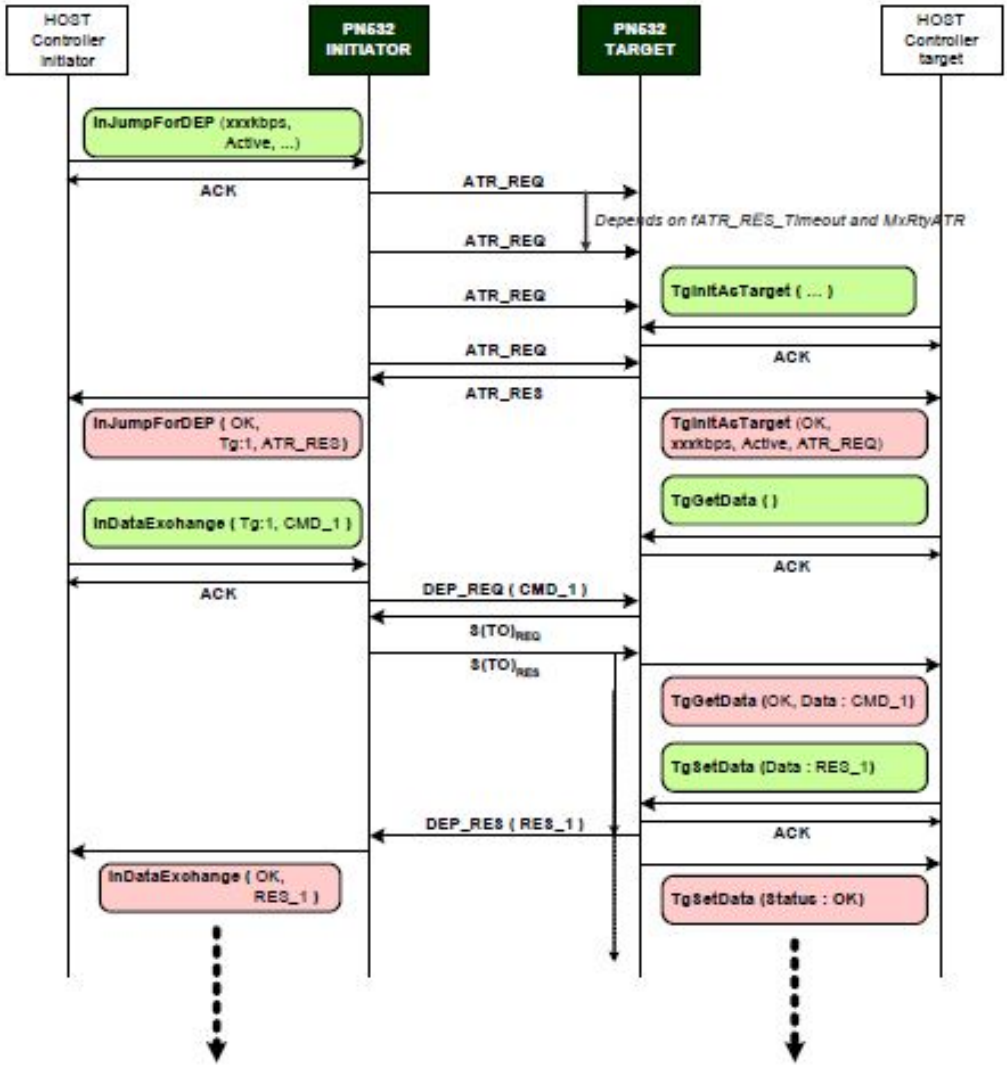


Figure 4.4: Time-sequence diagram of a peer-to-peer communication in active mode between two PN532 devices. From the PN532 User Manual [17]. The commands are in green and the corresponding responses are in red.

In the diagram, we see that the *PN532* must receive a series of commands from the host controller (in green on the diagram). It then directly sends back an *ack* message to the host controller to acknowledge good reception of the command. Finally, when the command has been executed, it returns a response message to the host controller (in red on the diagram). This response message may, for example, contain a status or the message received from the other device.

In our case, these commands, responses and *ack* messages are sent over the I2C bus. Thus, the port controller and the TW controller must be properly locked and unlocked as shown in listing 4.5. Sending a command simply consists in a series of calls to the `Write` method of the TW controller and receiving a response or an *ack* message consists in a series of calls to the `Read` method. Those action are implemented respectively in the `write_cmt` and the `read_dt` private methods. But the `read_dt` method is a bit more particular. Indeed, as shown in the diagram, the *PN532* may not respond rightaway to a command because it must wait for a response from the other NFC device. Therefore, the `read_dt` method uses a *loop* to try twenty-five times to receive the response. In our implementation, after each trial, the task is put to sleep for ten milliseconds using the `sleep` method of the `System` instance. Doing so, it allows other tasks to run. The resources are unlocked just before this instruction so it even allows another NFC communication to take place on another face. However, the NFC controller corresponding to the current face must stay locked during the whole NFC communication. So it is an example here of replacing a *busy-wait loop* with something more efficient thanks to the multitasking software.

It is the role of the NFC controller to ensure correct parameters are used for these commands and to verify proper reception of the *ack* messages and responses. To do so, we encapsulate the whole process into two functions: `Send` and `Receive`. In our implementation, the response of the target is fixed and contains only the letter T. But in the future, it can be changed to make the target answer accordingly to what it has received.

Additionally, the *PN532* provides features for battery savings. Instead of keeping it always activated, we can use the command `POWERDOWN` to put it in "power down mode", a battery-saving mode where most of its elements are turned off. We do it at the end of a NFC communication for both the *target* and the *initiator*. By sending the correct parameters with this command, we can configure the PN532 to wake up upon detection of a RF field or upon reception of a message on the I2C bus. Moreover, it will send an interrupt when wakening up. This interrupt will be handled by the port controller (see Section 4.2.5).

However, this working process may be problematic. It works well as long as the *target* is turned off after the *initiator*. But otherwise, it causes severe errors. This problem is explained in section 5.1.3.

4.3 The Main File

Finally, the behaviour of the software is defined in the *main.cpp* file. The user is strongly advised to modify this file and this file only. We have designed the overall system by trying to "hide" as much as possible complex mechanisms in controller's public methods such as `Send`, `Receive`, `SetAllColorsOnFace`, etc.

From this file, the user can access the instances of the `System` object and of each of the controllers by using the methods `instance` or `GetInstance` as shown in listing 4.7.

```
System::instance()
CTimer::instance()
CHUARTController::instance()
CPortController::instance()
CTWController::GetInstance()
//not implemented yet for nfc and led...
```

Listing 4.7: How to access the instances of the `System`, the timer and the controllers.

In the *main.cpp* file, the user can define and schedule the tasks simply by defining a function and using `schedule_task`. If needed, the function can also be member of a class, but then it needs to be declared *static*. In any case, the function itself must respect the following constraints:

- It must call the function `exit_task` in order to terminate. Calling `return` instead will result in unexpected behaviour,
- It can take only one argument in the form of a void pointer and it must manage itself the number and the types of the content of this argument,
- It must be resistant to the loss of its argument (see Section 5.1.5),
- It must have `void` as a return value,
- If it uses some critical resources, it must properly lock and unlock them and not keep them for too long (see Section 4.1.9),
- Its RAM usage must stay minimal in order to avoid overwriting other tasks' RAM spaces as there is no mechanism preventing the task from doing so (see Section 5.1.2).

The current main file can be found in the *Github* repository of the project [2]. It provides several tasks. These tasks serve as examples and were used to test the functioning of the software. Some functions are tasks and some are not. Tasks are recognizable because they use the method `exit_task` at the end. This may make the file unclear but we were incited to do so because we can run only three tasks at a time (see Section 5.1.1). So for testing, we often had to transform a task into a function and inversely. The proposed tasks are listed hereunder :

- `TestAccelerometer` prints the values obtained from the accelerometer, i.e. the accelerations on all three axes and the temperature,
- `InitPortController`, `InitLEDs` and `InitNFC` respectively initialized the port controller, the LED controller and the NFC controller, `InitPortController` must happen before any use of the port controller, thus before the two other init methods, as it also lists which faces are connected, and no TW message should be sent to an unconnected face,
- `VariateLEDsOnPort` takes an `EPort` as argument. It makes the light on that port flicker in a certain colour forever,
- `dummy5` prints the time every second and forever,

[2] `https://github.com/romische/Smartblocks-os`

- `LEDtask` initializes the port and led controller, then makes two faces flicker by launching two times the `VariateLEDsOnPort` method,
- `LEDoff` initializes the port and led controller. Just initializing the led controller turns them off,
- `InteractiveMode` continuously check the HUART for inputs from the user and the port controller for interrupts signalling an incoming message, then acts as needed,
- `NFCtask` initializes the port, led and NFC controller. Then launch two `InteractiveMode` tasks.

Part 5

Results

This section presents the results obtained with the work described in section 4. As previously said, the software is not perfect and contains a number of significant flaws. In fine, the main result of our work is the highlight of those errors, as they are important, complex and determinant.

In the first section 5.1, we explain their causes and effects and propose solutions. Then, section 5.2 presents the results obtained in term of performance and section 5.3 presents with pictures the experiments that were made. Finally, we advise future work directions in section 5.4.

5.1 Known limitations and bugs

In this section, we present the eight deficiencies that our software features, by developing for each: severity, symptoms, consequence, cause and several solutions. The four first problems are really important because they are critical. Then the next four are relatively easy to correct and may not have severe consequences.

5.1.1 RAM Division Problem

Severity: critical

Symptoms: A lot of tasks get refused by the system. Correct error handling must be programmed to see this. The user can use the fact that the method `schedule_task` returns the value -1 when there is not enough RAM to accept a new task.

Consequence: As we explained at the beginning of section 4, our motivation for our project is, principally, to increase communication speed. To do so, we envision to enhance the blocks with the ability to do several tasks simultaneously. For example, we can imagine a task that continuously checks for incoming messages. Then, when a message is received, it may trigger some actions that the block must do. Typically, when we imagine a large structure where many messages are passed in every direction, one example of such action is to send a message on each of its faces. To do so, the ideal scheme would be to create six tasks, each handling an NFC communication on one particular face. But, because

of the division RAM problem, it is not possible. Indeed, we have a limit of 3 tasks that can run concurrently at once. Therefore, if we try to implement this behaviour on the smartblocks, we will observe that many tasks are refused by the system. In the worst case, they are discarded and the desired actions never actually happen. In the best case, we program the smartblocks to continuously try until the task is accepted by the system, but in the meantime, a lot of computing power is consumed for nothing, and communication process becomes eventually even less efficient than with a single-task system. So this is one example of a situation where the RAM division problem causes severe limitations to our software, but we can easily show that it will be the case in many other situations.

Cause: As stated before, the ATMega328p provides a small RAM memory of 2 KB (2048 bytes). Such a small memory is difficult to cope with. The first, unavoidable problem is that a small memory means a small number of tasks that can run concurrently. The section 4.1.5 explains how to choose correct values to find a good balance between the size of the stack dedicated to one task, and the number of tasks. In our case, we get the values of 3 tasks each having 256 bytes of RAM at disposal. We then stated that when the user chooses a stack size too small, it will cause errors (the RAM overwrite error, in the next section 5.1.2). And when he chooses a small number of tasks running, it is not practical for the application we envisioned for the smartblocks. This latter problem is what we explain in this section.

Solution 1: Design the distribution of tasks by keeping this in mind. This is what is done at present. This could be achieved easily by replacing the call to `schedule_task` by a series of instructions stating: "`if` enough room, run as a task. `else` run as part of the current task."

Solution 2: By understanding what kinds of data are stored in sections *text*, *data*, *bss*, and on the heap, the user can try to change the code so that fewer data are stored in these sections. For example, typical RAM consuming data are definitions of strings for debugging. A solution would then be to remove them all. Alternatively, there exists a way to store the strings in the flash memory instead of the RAM.

Solution 3: Gain some RAM space by slicing the RAM differently, with blocks of heterogeneous length. In this solution, the user must first define a set of small tasks that will, in any case, use the same amount of RAM. For this, he can use the techniques shown in 4.1.5 to fetch the value of the stack pointer at several points of execution of the task. Once he gets the value, in bytes, of the RAM space needed for correct execution, he needs to create an object "task" that contains this value, the definition of the task (as a lambda function), an id, a constructor, and finally the parameters we can find in the structure `task_definition` (see section 4.1.6). Finally, he must considerably change the functioning of task creation and RAM space allocation. The goal is to arrange the RAM stacks dynamically, as shown in Figure 5.1. This solution is not easy to implement, but the user can base his work on the implementation presented in this master thesis.

Solution 4: Finally, some of the issues caused by this "RAM Division Problem" could be solved by implementing priorities into the task. The idea is that task with low priority

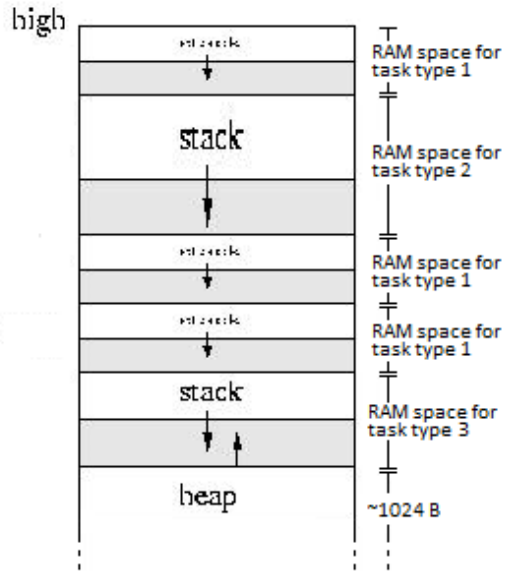


Figure 5.1: Allocation of the RAM when implementing the idea of tasks using a fixed amount of RAM space.

could be stopped to be replaced by tasks with low priority. Low priority can be used for tasks that can be stopped and restarted while losing all their data.

5.1.2 Stack Overflow Error

Severity: critical

Symptoms: When this error happens, the system indefinitely restarts. As for the *stack overflow* error, this problem is quite tricky to spot, because it can happen after a consequent, sometimes random, amount of time.

Consequence: Complete crash of the system. The smartblocks is unusable until you reset it.

Cause: The *stack overflow* problem can occur in many microcontrollers. Typically, this problem occurs when both the stack and the heap grow too much (explanation about the role of the stack and the heap can be found in section 4.1.5). As illustrated in Figure 5.2, it causes the heap and the stack to query the same address in the RAM, and as no mechanism prevent this from happening, one will write over the data of the other. This cause a loss of information that is in most of the case critical: the whole system crashes. One common cause, the one illustrated in Figure 5.2, is an excessive fragmentation of the heap.

In our case, we do not use the heap, but we have multiple and smaller stacks each dedicated to one task. If a task needs more RAM space than `TASK_STACK_SIZE` (here 256)

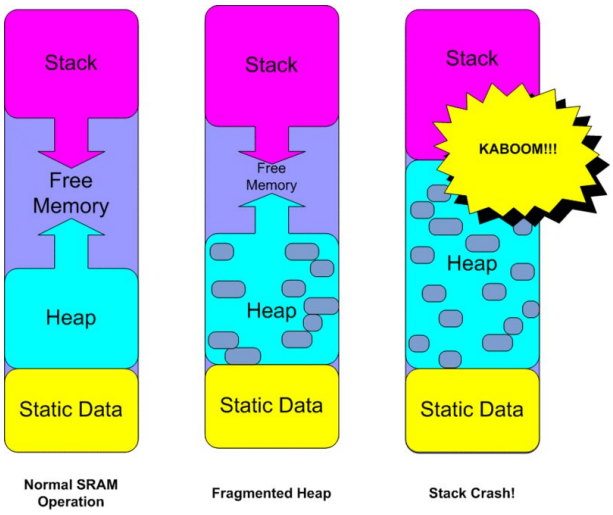


Figure 5.2: A schema showing the process leading to the *stack overflow* error.

bytes, it will start to request RAM addresses that were originally reserved for another task, thus causing a stack overwriting. As we have a greater number of tasks, we have a greater chance to observe a stack overwriting between two tasks.

Moreover, as explained in section 4.1.5, the allocation of RAM space for the main loop is not managed optimally. It implies that the user must keep the code before the run method as simple as possible. If he does not do it, we will observe that the main loop overflows the ram space of the first task. Typically, the problem will then arise when calling the `run` method, who directly starts the first task.

Solution 1: When this happens, the user must widen the space reserved for one task by changing the value of the `TASK_STACK_SIZE` (and consequently the value of `MAX_TASKS`). But on the other hand, doing so will exacerbate the RAM division problem (5.1.1), so this solution is not optimal.

Solution 2: Knowing that we can fetch the stack pointer value at any time, as explained in section 4.1.5, it would be easy to implement a mechanism preventing this from happening. The biggest difficulty is to find the optimal way to handle the error. Some ideas are : discard the task, discard the one of the two task that has the lowest priority (if priorities are implemented), reserve another portion of the RAM as a "buffer", intended to be used by tasks that need more RAM space than what they were given, and, finally, reuse addresses at the beginning of the RAM portion dedicated to the task, as it is done when using a "ring buffer". However, this solution would require either to constantly verify the value of the stack pointer, either to modify the C++ runtime itself.

Solution 3: If the user can implement tasks that use a constant amount of RAM in every case, then this problem will not happen anymore. Additionally, he will be able to solve the RAM division problem (5.1.1) with the proposed solution 3.

5.1.3 Time uncertainty for NFC Exchange

Severity: one critical and one moderate

Symptoms:

- **Critical error** NFC communication from the *initiator* block to the *target* block does not work anymore. It starts to work again after we successfully do the reverse communication.
- **Non-critical error** A lot of NFC exchanges fail. It may never happen if the user manages to create a system where all blocks always have the same amount of tasks running and getting the same results.

Consequence:

- **Critical error** The face of the target block does not anymore send an interrupt upon sensing of an RF field. In the current implementation, it implies that the face is not able anymore to receive a message. However, wake up on I2C message still works. So we can "reset" the face by sending a simple `POWERDOWN` instruction, that will wake it up then power it down correctly.
- **Non-critical error** As a lot of NFC exchanges fail, and it causing the overall system to be slower than expected. It is quite important as the goal of the project was to get a more efficient and rapid system.

Cause: We have no guarantees about the value of the time interval between the moment when a task is interrupted and when it is restarted. This depends on the number and type of the other tasks that are running. At the worst case, where all other tasks run without interrupting themselves (with `yield`, `sleep` or a refused `lock`), a task has to wait for one full turn of the round-robin scheduling before getting its execution time again. It means an additional delay of $(number\ of\ tasks - 1) * tick\ interval$ milliseconds. In our case, it is equal to $(3 - 1) * 10 = 20$ milliseconds. But it can be even worse if we take into account the use of lockable resources. Indeed, if such a resource is needed for the task to continue its execution, it may happen that it was locked by another task just before. Thus the task is blocked and has to wait for an additional turn of Round-Robin for 20 supplementary milliseconds, eventually multiple times. This behaviour is unpredictable but could be limited by choosing another lock system (see section 5.1.6). And at best, if there are no other tasks running or if all other tasks directly suspend themselves, there is no additional delay.

As two blocks may have a different amount and types of tasks running at a certain moment, the time interval between the interruption of a task and its restart may be different for each block. When we talk about two neighbouring blocks, this time uncertainty cause errors during NFC communications. An NFC exchange is successful if it happens as shown in the timing diagram 5.3. This exchange is also explained in section 4.2.7.

If one of the two neighbouring blocks gets delayed during the transaction, it will cause the loss of the message. An example of such situation is shown in Figure 5.4, but other chains of events can cause this error. However, this is not yet critical because the system must be capable to handle a failed transaction, as it can happen for other reasons (failure of the NFC chip, failure of the I2C bus, etc). Nevertheless, if this problem is not solved, NFC errors caused by the time uncertainty will occur more often, causing the overall system to be slower than expected.

Finally, a critical error occurs if the target block gets hugely delayed between the call to `P2PTargetTxRx` and the call to `PowerDown`. This causes the target to power down after the initiator does. We know the effect of this error but we did not find any explanation in the PN532 User Manual [17]. A discussion was started in the NXP community forum [1] to learn more about this issue, but, at the time of writing, it has got no answer yet.

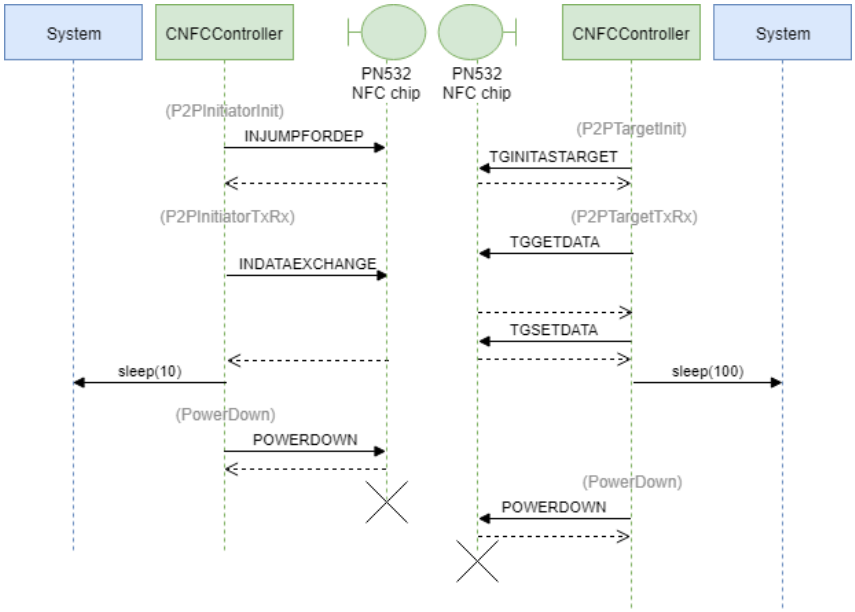


Figure 5.3: Time-sequence diagram of a successful NFC communication.

Solution 1: Wait for a sufficiently big amount of time between the end of the transaction and the `POWERDOWN` instruction. This is the solution that is currently implemented, as the initiator waits for 10ms, and the target for 140ms. Those values were found by trial and error.

Solution 2: Wait until the RF field disappears before using the `POWERDOWN` instruction. For this, we can check if the NFC chip senses an IR field by checking the status with the command `GETGENERALSTATUS`. We tried to implement this solution (see method `HasExternalField` of the `CNFCController` object), but without success and we don't know the reason.

Solution 3: Find a way to spot that the problem happened - but we did not have suggestions for that - and solve it by simply sending the `POWERDOWN` instruction.

[1] `https://community.nxp.com/thread/475204`

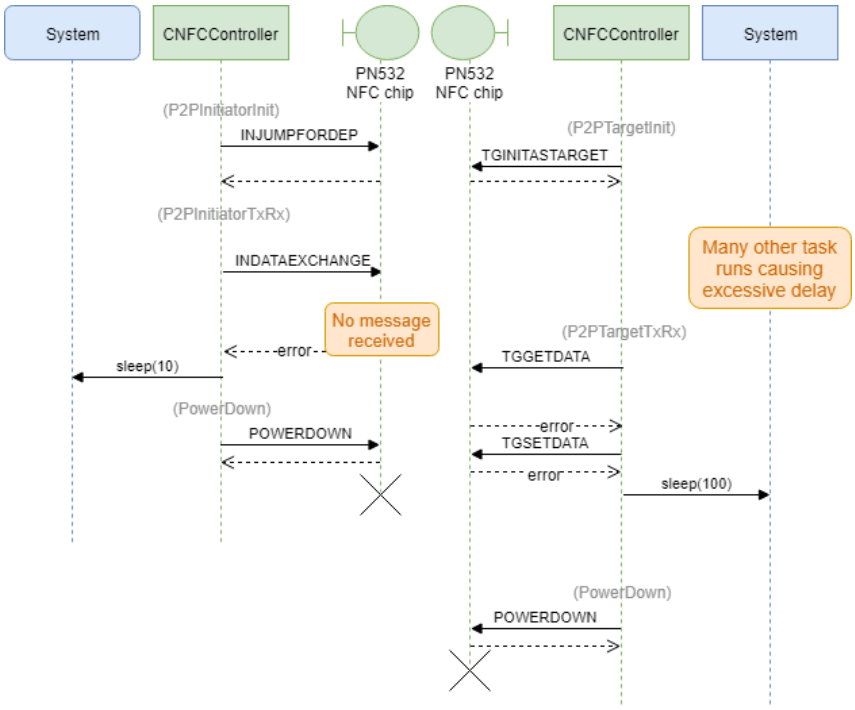


Figure 5.4: Time-sequence diagram of an unsuccessful NFC communication.

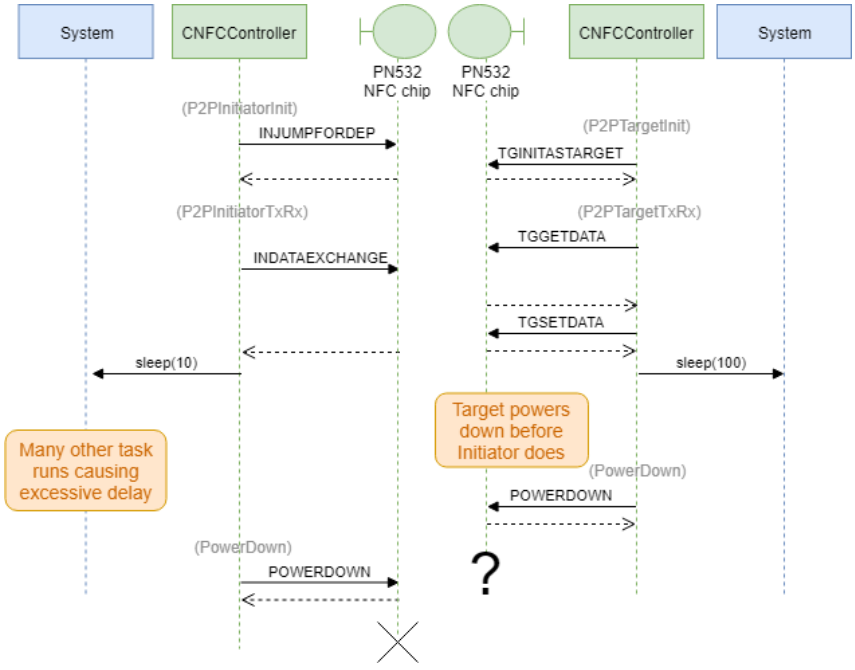


Figure 5.5: The time-sequence diagram of an unsuccessful NFC communication that causes further communication to be impossible. This happen when the target powers down after the initiator does.

Solution 4: Maybe use another message exchange scheme. Several different way to communicate are proposed in the *PN532* User Manual [17]. One idea is to configure the target in passive mode instead of the active one as implemented presently.

Solution 5: In fact, the `TGINITASTARGET` powers down the NFC chip automatically if no RF field is sensed, and wakes it up when a message needs to be received. Therefore, we think that a similar sparing of energy would be achieved by having all faces always initiated as target, then initiate them as initiator when needed.

Solution 6: The problem would be solved by stopping the task switching during the period of an NFC transaction. If we plan to do so, we might prefer to implement a cooperative multitasking system (see section 4.1.1). However, the NFC transactions will then be done on one face at a time, and because we could not anymore reuse the time taken by the frequent busy-wait loop to execute another task. Thus this solution would cancel the main benefit of using a multitasking system.

5.1.4 Blocking Eternal Loop in the TW Controller

Severity: critical

Symptoms: After a certain amount of time (random), the software stops working. If the LEDs were flickering, they get paused. If one neighbouring block sends an NFC message, it gets no response. And we cannot send or receive messages via the HUART controller as it stops working too.

Consequence: Complete crash of the system. The smartblock is unusable until you reset it, disconnect it and remove the battery.

Cause: As explained in section 4.2.4, the TW controller is driven by an interrupt. This interrupt triggers the "state" to change, the error code to correspond to whatever error happened, and the messages to be sent and received. We suspect that sometimes this interrupt gets discarded because it happened while the system was in a "`cli-sei`" period (see section 4.1.1). Yet, in both the methods `Read` and `EndTransmission`, the TW controller use an eternal loop to wait for the state to change from `PN532_I2C_BUSY` to `TW_STATE_READY`. The code coming afterwards can only continue if the state changed correctly. In our case, it will block everything that needs the I2C bus to function, because the TW controller was *locked* before this error happened (see 4.1.9). However, we observe that the HUART controller is blocked too, and that is quite unexpected because this object does not use the TW controller. Therefore, we are totally uncertain about the cause of this error as we explained it here. However, in the case we were not wrong, we still provide potential solutions hereunder.

```
 // wait until twi is ready, become master transmitter
while(unState != TW_STATE_READY) {
   continue; // os sleep
}
```

Solution 1: Simply put a timeout on the eternal loop. Or, alternatively, a timeout to the lock. This could be more interesting because it would prevent other similar kinds of errors, but it is way more difficult and resource-consuming than the timeout in the TW controller that would simply consist in using a `for` loop instead of a `while` loop.

Solution 2: If the cause of the error is not what we explained here, further investigations will be required. To debug it easily, we advise using a logic analyser, which serve to peek the value of several pins, and which could be used here to watch what happens to the expected interrupt.

5.1.5 Disappearance of the argument of a task

Severity: moderate

Symptoms: The argument of a task suddenly change its value.

Consequence: It may cause serious execution errors, as an action that is executed on the wrong face, a LED that is lit up with the wrong colour, etc.

Cause: In the current implementation, when passing an argument to a task, we store it on the stack with a simple declaration instruction, then we pass its pointer (its address) to the `schedule_task` method. Doing so, we observe that it may happen that the argument gets replaced by another random value.

Solution 1: We think that changing the classic variable to a `constant` variable could solve the problem.

Solution 2: The first action of a task must be to store the *value* of its argument onto its stack. This solution is not ideal as it may happen that the value of the argument get changed even before the task has gotten some execution time.

Solution 3: The problem will disappear if we implement a class describing a task that stores the arguments in its class variables.

5.1.6 Unfairness of the lock

Severity: moderate

Symptoms: When we design a task that continuously locks and unlocks a resource, we observe that this one gets all the execution and the other tasks, if using the same resource, seem to be disabled.

Consequence: One or several tasks not running.

Cause: The locking mechanism is explained in section 4.1.9. With the present implementation, if a task continuously locks and unlocks a resource, and in particular if it does nothing between unlocking and locking (meaning the locking instruction occurs just after the unlocking one), it is probable that the context switch will always occur when this resource is locked. Therefore, the other tasks trying to lock that resource will never get it, and will never continue their execution farther than that.

Solution 1: What is done is the current implementation is that we ensure such tasks `yield` at some point. This is the same concept as a cooperative multitasking system (see section 4.1.1). This is because of this easy solution that we classified this error in "moderate".

Solution 2: This would be solved simply by using a better locking mechanism. There are several kinds of locks proposed in the literature to solve the problem of concurrent access. Here we use a very basic one, called *spinlock*. Another implementation was proposed in *avr-os*, that it called *mutex*. It functions similarly, except that a task that was provided access to a resource can record itself as "the next one". Upon unlocking of the resource, the lock will be given to this task only. Along with a round-robin scheduling, this ensures fairness of the lock, as each task get the chance to "reserve" the resource locked by the task running just before in the circular order. We did not implement it here, as we thought that this implementation doesn't improve much the software, while it uses a bit more of memory. However, we were proved wrong, as shown in experiment 5.3.1.

5.1.7 Imprecision of the sleep function

Severity: moderate

Symptoms: When calling the method `sleep`, a task gets delayed less or more time than expected.

Consequence: Actions are slower than expected. For example, when lighting the four LEDs of one face, we send many TW messages. Each TW message is preceded by a call to `SelectPort` and therefore a call to `sleep(1)`. If all those `sleep(1)` acts like a `sleep(10)`, we can observe the delay with the naked eye. It may also cause problems for communication over NFC, which is dependent on a certain synchronisation between the sender and the receiver, as explained in section 5.1.3.

Cause: The `sleep` method has been explained in section 4.1.8. This method is imprecise for two reasons. First, because the sleep values of every task get updated every 10ms by the interrupt. After the call to this method, the interrupt can occur after a variable time interval between 0 and 10ms. Secondly, because the task's `sleep_time` attribute, that stores how many milliseconds left the task needs to sleep, gets reduced by constant steps of 10 (or the value of the `TICK_INTERVAL` macro), not by the exact time that passed that could be fetched from the `timer` object. This means, for example, that calling `sleep(1)` is equivalent to `sleep(3)`, `sleep(6)` and even `sleep(10)`, as all these values will equivalently trigger the task to regain execution state after the next interrupt. And as this interrupt can happen after a time varying between 0 and 10 ms, then all the calls to `sleep` with a value between 1 and 10 will result in a sleep between 1 and 10 milliseconds. Same applies for `sleep(23)` and `sleep(29)`, etc.

Solution 1: Use `sleep_time` to store the time in milliseconds when the task can regain the running state. Then, in the interrupt, we check if the value of the timer is greater than `sleep_time`. This solves just one part of the problem, as the verification will still occur every 10ms.

Solution 2: In the software *avr-os*, decrementing the `sleep_time` value was the role of a task created by the system at the beginning and not the role of the interrupt routine. And additionally, it was decreasing the value by the exact time, and not a fixed step as here. This solves the error as we are guaranteed that this task will run regularly, thus the `sleep_time` value will be decremented with more precision.

Solution 3: The sleep could be implemented similarly to a lock, i.e. with an eternal loop constantly checking for the timer. The loop stops if the value of the timer indicates that the task can restart, and otherwise it yields and continues. This solution is ideal as it both spares some instructions in the interrupt routine and the creation of one additional task. The computation needed for the sleep is in fact done during the execution time of the task itself.

5.1.8 Restart of the timer

Severity: moderate

Symptoms: After a certain time, the timer gets restarted.

Consequence: For every use of the smartblocks were timing is important, it could cause errors. However, such restart is inevitable, because we will always have a limit to the millisecond count caused by the finite storage space we use. Therefore, the system must be able to cope with such restarts in any case.

Cause: The first suspected cause is of course that the counter (variable `m_unTimerMilliseconds`) as reached its limit and that the instruction `m_unTimerMilliseconds++` cause the counter to become 0. However, in our case, we used an `uint32_t` integer, meaning the maximum reachable number is $2^{32} = 4294967296$ milliseconds, that corresponds roughly 1193 hours. Yet, we observed the error after a time that was much smaller. This is caused by unpredictable memory corruption.

```
volatile uint32_t m_unTimerMilliseconds;
```

Solution: However, another implementation of the timer was provided by M. Allwright. This implementation might not suffer from this error so we would suggest replacing the `CTimer` object by this implementation.

5.2 Performances

In order to get an idea of the performances achieved, we measure the time needed by the smartblocks to perform some key actions, both in the case of single-tasking and in the case of multitasking. We expect the multitasking software to be a bit slower at accomplishing those basic actions, as tasks have to share the execution time.

Table 5.1 shows our results. They were computed only twice, with the exact same software. Hence they may not be very accurate but we still think they are representative of the general case. The values for multitasking were computed with no other task running, so, if we observe a delay, it is the delay due to the execution time taken by the context switch and the lock-unlock actions only. However, the differences can also come from other more meaningless differences between the single-tasking and the multitasking software. Additionally, the receiving of an NFC message in the case of multitasking requires an additional delay of 100ms before powering down, as explained in section 5.1.3.

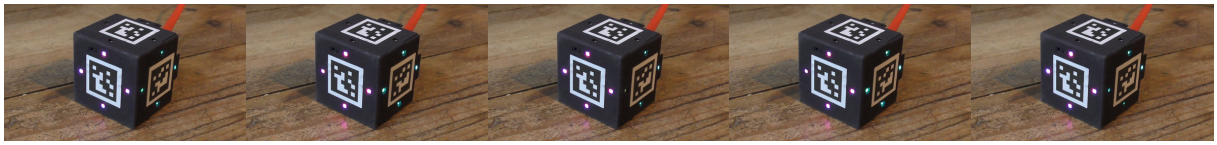
	single-tasking	multitasking
lighning the four LEDs of a face	4-6ms	5ms
failed to send a nfc message	379-381ms	482-483ms
succeed to send a nfc message	244-262ms	623-624ms
fail to receive a nfc message	540-542ms	577-578ms
succeed to receive nfc message	150-172ms	608-609ms

Table 5.1: Comparison of the time interval needed by both the single-tasking software and the multitasking software to perform some key basic actions.

As expected, the multitasking software is slower in most of the cases. However, while it is slower for basic actions, it has the potential to increase the speed of message passing, and thus reduce errors in the constructing process.

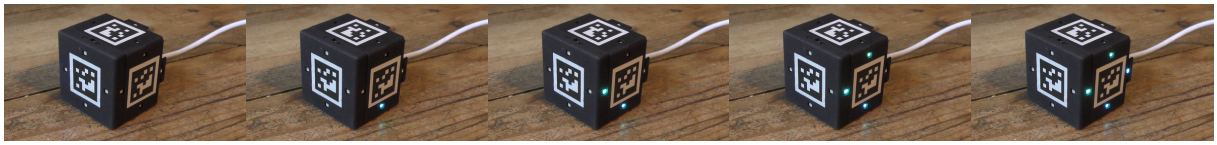
5.3 Experiments

We designed four experiments to test the capabilities of our smartblocks and compare the efficiency of multitasking versus multitasking. All the experiments were filmed with a slow-motion camera at 100fps, meaning one image every 20ms. We show here only the frames that are relevant.

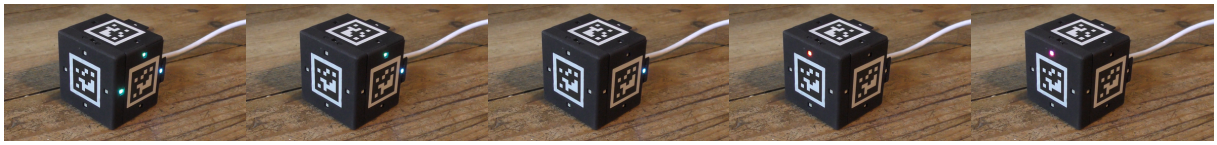


(a) t=0ms (b) t=20ms (c) t=40ms (d) t=60ms (e) t=80ms

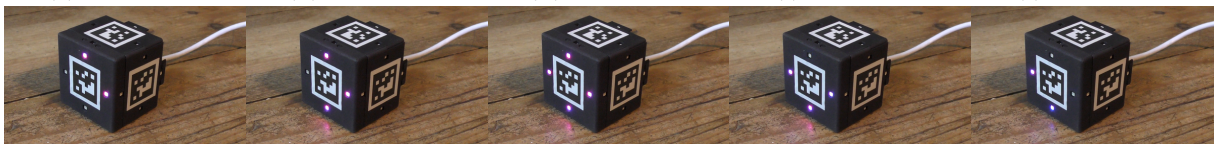
Figure 5.6: Blinking LEDs on two sides in the single-tasking environment



(a) t=0ms (b) t=60ms (c) t=140ms (d) t=200ms (e) t=260ms

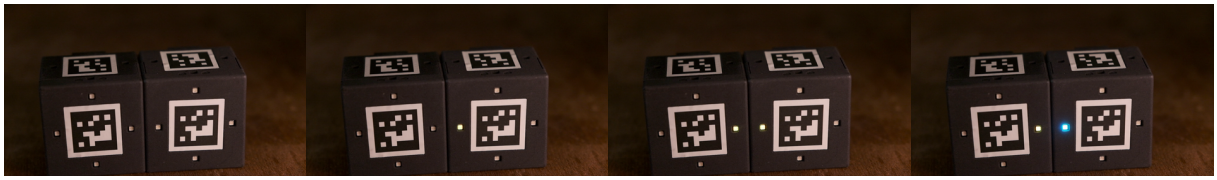


(f) t=320ms (g) t=380ms (h) t=440ms (i) t=500ms (j) t=540ms

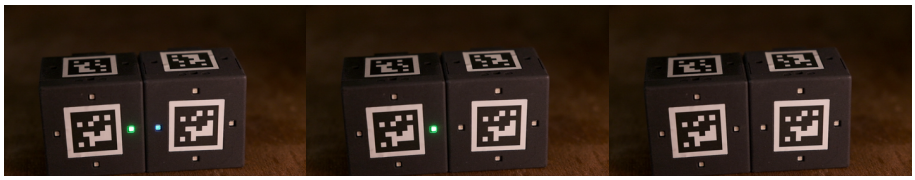


(k) t=600ms (l) t=660ms (m) t=720ms (n) t=800ms (o) t=860ms

Figure 5.7: Blinking LEDs on two sides in the multitasking environment

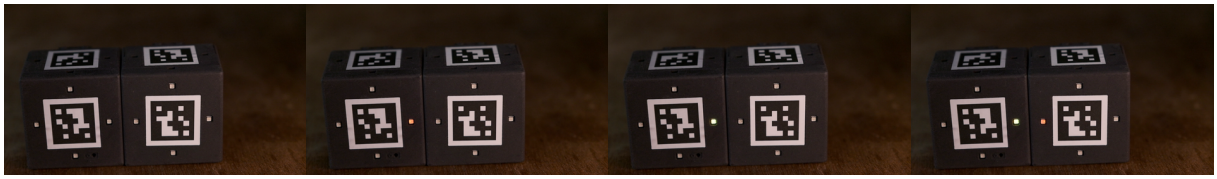


(a) t=0ms (b) t=10ms (c) t=30ms (d) t=260ms

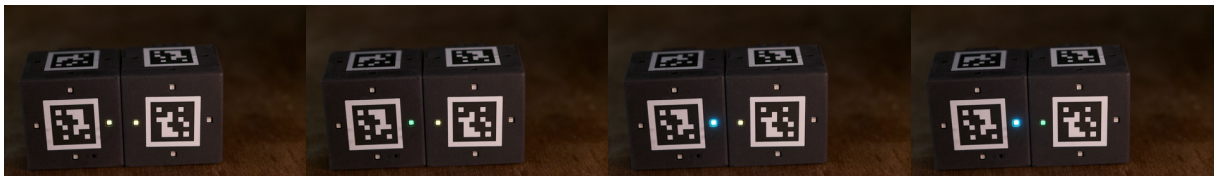


(e) t=360ms (f) t=370ms (g) t=470ms

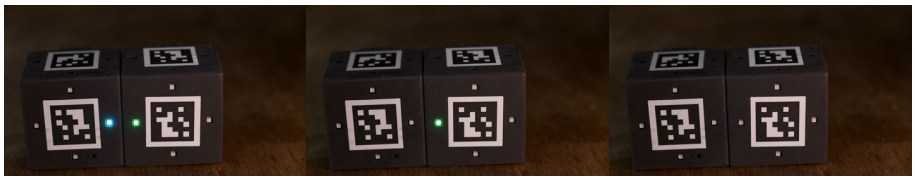
Figure 5.8: A NFC transmission in the single-tasking system



(a) t=0ms (b) t=10ms (c) t=40ms (d) t=200ms

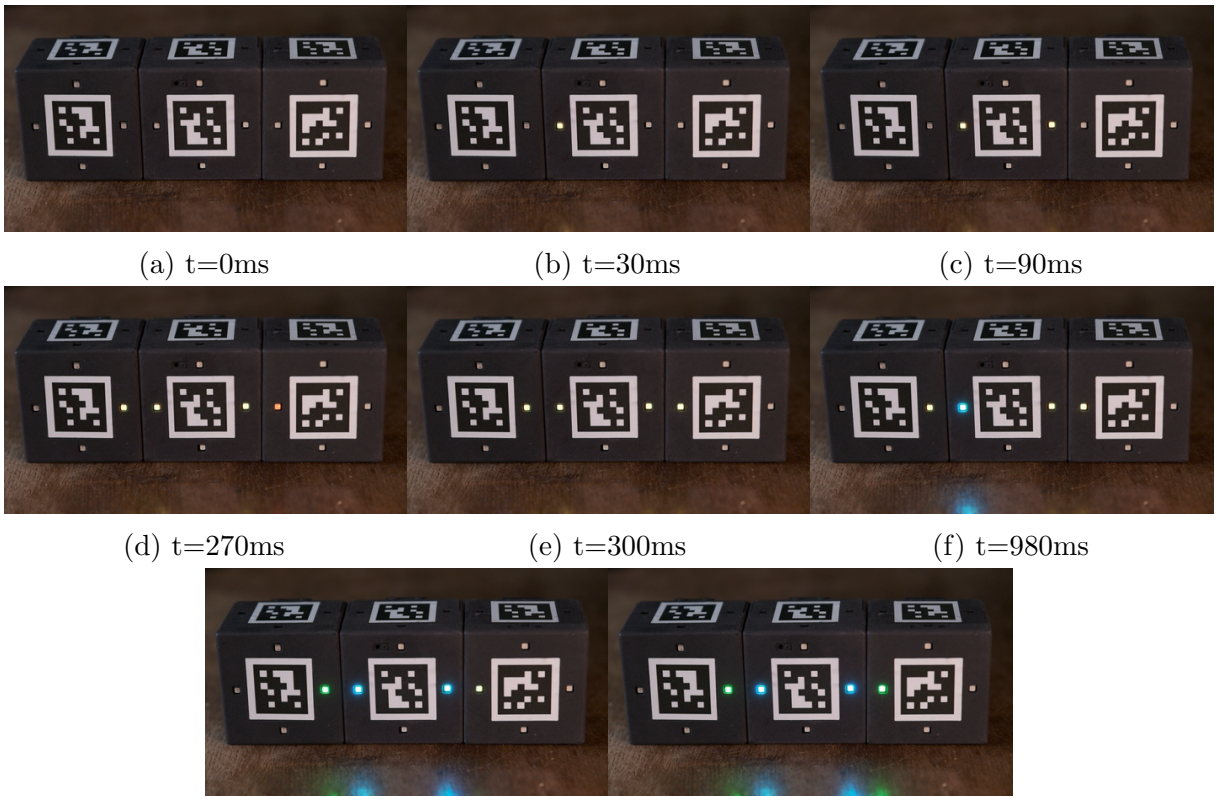


(e) t=230ms (f) t=910ms (g) t=950ms (h) t=1000ms



(i) t=1020ms (j) t=1110ms (k) t=1180ms

Figure 5.9: A NFC transmission in the multitasking system

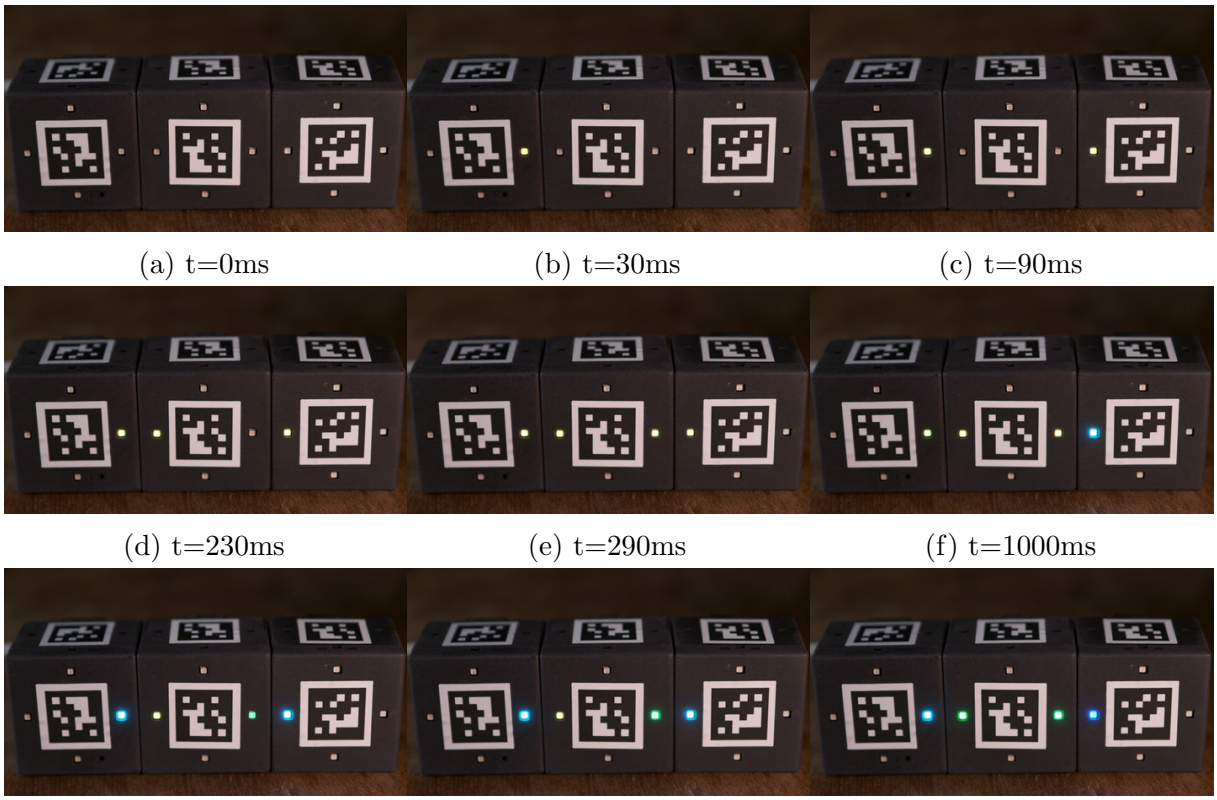


(a) t=0ms (b) t=30ms (c) t=90ms

(d) t=270ms (e) t=300ms (f) t=980ms

(g) t=1060ms (h) t=1090ms

Figure 5.10: One block simultaneously sending two NFC messages



(a) t=0ms (b) t=30ms (c) t=90ms

(d) t=230ms (e) t=290ms (f) t=1000ms

(g) t=1050ms (h) t=1070ms (i) t=1150ms

Figure 5.11: One block simultaneously receiving two NFC messages

5.3.1 Singletasking vs. Multitasking: Blinking LEDs on Two Sides

The following listings 5.1 and 5.2 show the algorithms used for blinking LEDs on two different faces in two different colours, respectively for the single-tasking environment and for the multitasking environment. Figures 5.6 and 5.7 show these algorithms in action.

```
loop forever :
   select port WEST
   wait for 1ms
   light all four LEDs in turquoise

   select port NORTH
   wait for 1ms
   light all four LEDs in pink

   select port WEST
   wait for 1ms
   turn off all four LEDs

   select port NORTH
   wait for 1ms
   turn off all four LEDs
```

Listing 5.1: Algorithm for blinking LEDs on two sides, in the single-tasking environment

```
define the blinking task on port X and color C:
   loop forever:
       light all four LEDs of port X in color C
       turn off all four LEDs of port X
       yield

schedule blinking task on WEST face, in turquoise.
schedule blinking task on BOTTOM face, in pink.
```

Listing 5.2: Algorithm for blinking LEDs on two sides, in the multitasking environment

In both cases, the lighting of the four LEDs is made one after another. That's what we expected to observe by capturing 50 pictures per second. However, the single-tasking system is so fast that we don't see it. It seems the light does not have the time to completely turn off before it is re-lighten up again.

For the multitasking environment, on the other hand, the call to `SelectPort` is made by the LED controller just before sending a TW message. As explained in section 4.2.5, this method must mandatorily be followed by a call to the `sleep` method for 1 millisecond, and as explained in section 4.1.8, such small delays have many chances to result in a 10-millisecond delay. As we light up four RGB LEDs, and we use one TW message by colour component, this results in the multitasking software being hugely delayed compared to the single-tasking one. Therefore, with the multitasking software, we can observe the LEDs being lit up one at a time.

When designing this experiment, we expected to observe a certain interleaving of the tasks by having one led lit up on one face, then one on the other face, etc. This does not happen here. However, other experiments where we provoked this interleaving with the `yield` method showed that it is possible and supported by the software. We suppose this is due to the default explained in section 5.1.6. This is indeed what happens if we remove the call to the `yield`. As the execution is always passed to the other task with the port / TW controller being locked, the other task never has the chance to execute. We solved this problem by adding the `yield`, but then the other task executes only when the execution is passed via this `yield`, thus when the other task has finished lightning and turning off its LEDs. That is why we observe one face being lit up at a time.

5.3.2 Singletasking vs. Multitasking: Sending an NFC Message

For this experiment, whose results are shown in Figure 5.8 and 5.9, we simply filmed one block sending an NFC message to another receiving one. Both blocks light up one LED of their BOTTOM face in yellow, at the beginning of the transaction. The LED they light up corresponds to the direction in which they transmit or receive the message. Then, if the transmission is successful, the sender lights this LED in turquoise, and in red if the transmission failed. The receiver does the same but in green and pink. As in fact, one LED is a pack of three LEDs (one red, one green and one blue), lighting a LED consists in lighting three LEDs.

The pseudo-code for the sender and the receiver with the single-tasking system are shown respectively in listings 5.3 and 5.4. For the multitasking software, shown in listings 5.5 and 5.6, we use the same course of actions, but with small differences :

- The NFC and LED controller must be properly locked and unlocked on the desired face,
- There is no need to select the port beforehand (and wait for 1ms) as it is done by the NFC/LED controller just before the sending of a TW message,
- The actions are launched as tasks.

```
define the sending port - WEST
define the corresponding led - 1

loop forever :
   //light in yellow
   select port BOTTOM
   wait for 1ms
   light LED 1 in yellow

   //send NFC
   select port WEST
   wait for 1ms
   send the NFC message

   //show the result during 100ms
```

```
    select port BOTTOM
    wait for 1ms
    if sending was successfull :
        light LED 1 in turquoise
    else :
        light LED 1 in red
    wait for 100ms

    //turn off light
    turn off LED 1

    wait for 900ms
```

Listing 5.3: Algorithm for sending an NFC message in the single-tasking environment

```
loop forever :
  check for input on the HUART
  /* check the value of this input
    and react if needed */
  else :
  check for interrupts on the port controller
    if it has one :

        foreach port X:
          if X corresponds to the interrupt :
            define the corresponding led - n

            //light in yellow
            select port BOTTOM
            wait for 1ms
            light LED n in yellow

            //send NFC
            select port X
            wait for 1ms
            listen for the NFC message

            //show the result
            select port BOTTOM
            wait for 1ms
            if receiving was successfull :
                light LED n in green
            else :
                light LED n in pink
            wait for 100ms

            //turn off light
            turn off LED n
```

Listing 5.4: Algorithm for receiving an NFC message in the single-tasking environment

```
argument - port X
define the corresponding led - n

loop forever :
```

```
    //light in yellow
    light LED n in yellow on BOTTOM

    //send NFC
    send the NFC message on port X

    //show the result during 100ms
    if sending was successfull :
        light LED n in turquoise on BOTTOM
    else :
        light LED n in red on BOTTOM
    sleep for 100ms

    //turn off light
    turn off LED n on BOTTOM

    sleep for 900ms
```

Listing 5.5: Algorithm for sending an NFC message in the multitasking environment

```
loop forever :
  check for input on the HUART
  /* check the value of this input
    and react if needed */
  else :
  check for interrupts on the port controller
    if it has one :

        foreach port X:
          if X corresponds to the interrupt :
            define the corresponding led - n

            //light in yellow
            light LED n in yellow on BOTTOM

            //send NFC
            listen for the NFC msg on face X

            //show the result
            if receiving was successfull :
                light LED n in green on BOTTOM
            else :
                light LED n in pink on BOTTOM
            sleep for 100ms

            //turn off light
            turn off LED n on BOTTOM
```

Listing 5.6: Algorithm for receiving an NFC message in the multitasking environment

In the multitasking environment, since the lightning of a LED is slower, we showed both the moment when the light starts to light and when is it completely lit up.

5.3.3 Sending two NFC Messages Simultaneously

For this experiment, we use the same algorithms that were presented in Figure 5.3 and 5.4. We simply program one block with two sending tasks, and surround it by two blocks programmed with the receiving task. Results are shown in Figure 5.10.

5.3.4 Receiving two NFC Messages Simultaneously

For this experiment, we use the same algorithms that were presented in Figure 5.3 and 5.4. We make the opposite of the previous experience: we program one block with two receiving tasks, and surround it with two blocks programmed with the sending task. In order to almost achieve simultaneous sending, we start the two blocks by pressing their reset button simultaneously. However, this synchronisation is not sufficient, as we can see a small delay in the pictures, are shown in Figure 5.11.

5.4 Future Work Directions

In this part, we explain the work that must still be done to make our system operational. The first step is, of course, to solve the four major problems and the minors one explained in section 5.1, by implementing their corresponding solutions.

Another work that must be done is to re-arrange the controllers to make them more practical to use for a programmer. Indeed, in the current implementation, the main file must respect many constraints, and a new programmer, if it has not read this master thesis, would easily make critical mistakes. Here we present several ideas to make the software more easy to use :

- Each controller should initialize itself seamlessly on the first use. In this way, we are certain that the initialization will be made in proper order, and additionally, the programmer does not need to worry about that.
- It must be the role of the port controller to keep a list of the connected faces and to update it from time to time.
- When calling `GetInterrupt`, the port controller should return the face on which the interrupt occurred instead of the code of the interrupt, that needs to be analysed afterwards.
- A lockable controller for the accelerometer should be provided.
- The LED and NFC controller should implement the multiton pattern from inside, and not in the main file as it is at present.
- The HUART controller should implement a method "print" that would directly print a full message along with the time.
- Some basis colours must be proposed by the LED controller.

- To gain efficiency, the port controller should keep track of which port is selected, and effectively change the port only if we ask for another one. In this way, we avoid many 1 milliseconds delays that are the main cause for the multitasking system to be slower than the single-tasking system.
- The tasks must be implemented as an object, to facilitate the tracking of the argument.
- When one task gets refused by the system, a method `schedule_task_or_do` should seamlessly launch the method defining it from inside the current task.

In addition, we also envision the development of some supplementary abilities that would be useful for a programmer :

- The ability to *kill* a task with its id. For this we must take care of freeing the resources locked by this task, so we need the `resource` class to keep track of which task owns it.
- A controller for the RAM, that has the ability to print RAM usage in order to take the decisions explained in section 4.1.5.
- The software should provide a default task that implements a simple routing algorithm, to pass simple instructions to one or several neighbouring blocks, or ask them to pass the instruction further.

Finally, we should analyse the gain of performance brought by the multitasking software in larger structures. For this, a basic routing algorithm must be implemented first. This routing algorithm will allow remote monitoring of the structure, which was the second, not achieved, objective of the project made as part of this master thesis.

Part 6

Conclusion

In section 2, we presented the concepts and motivations driving autonomous construction with swarm robotic, and explained the need for an efficient semi-active material. While such materials increase the cost of the system, we noted that using them brings significant benefits: First, the system becomes more robust and more efficient. Secondly, in some cases, adding additional block capabilities increase the number of feasible structures. Finally, a structure composed of intelligent blocks is an intelligent structure, and can be, for example, able to detect and react to hazards.

Then, in section 3, we reviewed and compared several implementations of such semi-active materials. We concluded that the Smartblocks are the most suitable for running experiments of autonomous construction with swarm robotics. Firstly because they are easy to manipulate and place, thereby we avoid complexifying the fabrication of the manipulating robots. Secondly, because they feature a mean of stigmergy more complete than other implementation: it allows per-face signals, thus enabling experiment with a series of algorithms that would not possible otherwise. Finally, in contrast with some other implementation, because they work in 3D structures.

After that, the section 4, we explained the functioning of our multitasking software, that was made for the smartblocks as the practical work of this master thesis. The software includes both a preemptive multitasking system, greatly inspired by an open-source software developed by Chris Moos and a set of controllers driving the hardware components of the Smartblocks.

Finally, in section 5, we explained what are the major flaws preventing our software to be fully effective, and their solutions. We also present some experiments and we also advise for future work directions, which fundamentally are to implement the proposed solutions and to implement a routing algorithm for testing in larger structures.

Our experiments show that simultaneous communication works sufficiently well when we do not encounter the issue of time uncertainty (see Section 5.1.3). While the fact of accomplishing several communications simultaneously makes the exchange a bit slower, multitasking should make the overall system more efficient, as it removes the delay caused by waiting for the completion of the NFC exchange on all other five faces that occurs

when single-tasking is used, and it removes a busy-wait loop that occurred very frequently. However, this has not been tested yet, as it is relevant only for large structures where many messages are passed in many directions.

Yet some of the limitations are critical cannot be left in a practical implementation. Some work is still required in order to use this multitasking software for real experiments. However, we think our work highlighted and contributed to explaining some obstacles that were not self-evident beforehand, and as such our will benefit the further development of a multitasking system for intelligent blocks.

Bibliography

[1] M. Allwright, *An autonomous multi-robot system for stigmergy-based construction*. PhD thesis, Paderborn University, 2017.

[2] J. L. Deneubourg, "Application de l'ordre par fluctuations a la description de certaines étapes de la construction du nid chez les termites," *Insectes Sociaux*, vol. 24, pp. 117–130, 1977.

[3] K. Sugawara and Y. Doi, "Collective construction of dynamic structure initiated by semi-active blocks," in *The Institute of Electrical and Electronics Engineers Inc. (IEEE) Conference Proceedings*, pp. 428–433, 2015.

[4] G. Theraulaz and E. Bonabeau, "Modelling the collective building of complex architectures in social insects with lattice swarms," *Journal of Theoretical Biology*, vol. 177, no. 4, p. 381–400, 1995.

[5] C. P. M. D. M. Allwright, N. Bhalla, "Towards autonomous construction using stigmergic block," Technical Report TR/IRIDIA/2017-003, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 2017.

[6] P.-P. Grassé, "La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs," *Insectes Sociaux*, vol. 6, pp. 41–80, March 1959.

[7] J. Werfel, D. Rus, Y. Bar-Yam, and R. Nagpal, "Distributed construction by mobile robots with enhanced building blocks," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2787–2794, 2006.

[8] G. Theraulaz and E. Bonabeau, "Coordination in distributed building," *Science*, vol. 269, p. 686–688, 1995.

[9] D. S. P. K. Eric Bonabeau, Sylvain Guérin and G. Theraulaz, "Three-dimensional architectures grown by simple 'stigmergic' agents," *BioSystems*, vol. 56.1, p. 13–32, 2000.

[10] D. Anderson, J. Frankel, J. Marks, D. Leigh, K. Ryall, E. Sullivan, and J. Yedidia, "Building virtual structures with physical blocks," in *Proceedings of the 1999 12th Annual ACM Symposium on User Interface Software and Technology (UIST '99)*, (Asheville, NC, USA), pp. 71–72, ACM, Nov. 1999.

[11] L. Buechley and M. Eisenberg, "Boda blocks: A collaborative tool for exploring tangible three-dimensional cellular automata," in *Computer-Supported Collaborative Learning Conference, CSCL*, vol. 8.1, pp. 99–101, 2007.

[12] B. C. Kirby, M. Ashley-Rollman, and S. Goldstein, "Blinky blocks: A physical ensemble programming platform," in *Conference on Human Factors in Computing Systems - Proceedings*, pp. 1111–1116, 2011.

[13] K. Gilpin, A. Knaian, and D. Rus, "Robot pebbles: One centimeter modules for programmable matter through self-disassembly," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2485–2492, 2010.

[14] K. Gilpin and D. Rus, "A distributed algorithm for 2d shape duplication with smart pebble robots," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3285–3292, 2012.

[15] J. W. Romanishin, K. Gilpin, and D. Rus, "M-blocks: Momentum-driven, magnetic modular robots," in *IEEE International Conference on Intelligent Robots and Systems*, pp. 4288–4295, 2013.

[16] J. W. Romanishin, K. Gilpin, S. Claici, and D. Rus, "3d m-blocks: Self-reconfiguring robots capable of locomotion via pivoting in three dimensions," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1925–1932, June 2015.

[17] NXP, *PN532 User Manual*. NXP. UM0701-022.