www.s2is.org
International Journal on Smart Sensing and Intelligent Systems
ISSN 1178-5608

OPERATION PRINCIPLE, ADVANCED PROCEDURES AND VALIDATION OF A NEW FLEX-SPI COMMUNICATION PROTOCOL FOR SMART IoT DEVICES

P. Visconti [1], G. Giannotta [2#], R. Brama [3#], P. Primiceri [4], R. de Fazio [5], A. Malvasi [6 #]
Department of Innovation Engineering, University of Salento, 73100, Lecce, Italy
[#] CMC Labs -A Division of CMC S.r.l.- C.da Pagliarulo sn – 72012, Carovigno (BR), Italy
Emails: paolo.visconti@unisalento.it[1], gmgiannotta@gmail.com[2], r.brama@cmclabs.com[3], patrizio.primiceri@unisalento.it[4]. defazio_roberto@libero.it[5], a.malvasi@cmclabs.com[6]

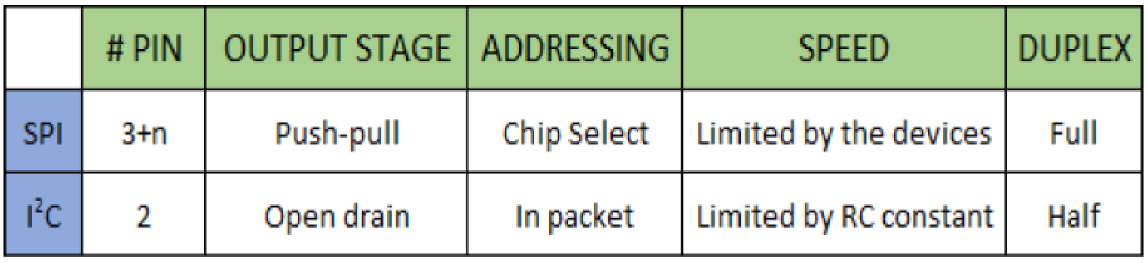
Submitted: May 25, 2017   Accepted: July 17, 2017   Published: Sep. 1, 2017

Abstract – ***In this paper, we report on hardware structure, operation mode and software development for a new advanced communication protocol whose aim is obtaining a fully shared SPI bus with a fixed amount of wires, without renouncing to advantages of a push-pull output stage and obtaining an architecture capable of great flexibility. All four signals of a classic SPI protocol are entirely shared by the slaves on bus: when a master wants to communicate with a particular device, it will perform an addressing at packet level: starting from its main characteristics, various adopted solutions to realize a shared SPI bus will be analyzed, explaining how a communication session is performed. The firmware structure was designed as a software stack composed by interacting layers, tracing model of similar protocols that share with FlexSPI some features. Some of the advanced procedures that can be performed thanks to this protocol will be discussed, highlighting the suitability of FlexSPI for dynamic smart objects; in fact, by adding these features to developed framework, it is possible to explore and appreciate expandability of this communication protocol, making it suitable to meet advanced IoT requirements of smart objects. FlexSPI can be built like a MAC layer above the SPI bus, to process all necessary pieces of information to perform the packet level addressing, using a stack having a layered architecture. This is idea followed in the firmware development, to implement this communication protocol, experimentally verified in the performed and reported communication tests, confirming that it is possible to obtain a shared push-pull bus.***

Index Terms: IoT, communication protocols, smart objects, FlexSPI, software procedures, shared bus, PIC.

I. INTRODUCTION

The critical overview on two of the most used communication protocols, SPI and I^2C, brings some considerations regarding how well they are suited for *Internet of Things* (IoT) applications [1]; moreover, this analysis can be used to understand what can be done to improve performances and possibly obtaining a new standard that combines the advantages of both of them. As observed, I^2C has many of the features required by the IoT paradigm, with an abstraction layer that provides flexibility and different services. However, although improved with the latest reviews, it still lacks of important features and, above all, its strict connection between number of devices and the maximum bit-rate forces to create restricted smart objects. At last, the purpose of limiting energy consumption fails due to the open-drain connection with pull-up resistor. On the other hand SPI, based on push-pull output stage, is a protocol less power consuming and with a throughput depending only on capabilities of devices on the bus. However, its simplicity causes the lack of features necessary to let subsystems communicate in an efficient way, making difficult its implementation in a smart object [2] [3]. The main features of these two protocols, compared in some of the most important aspects, are here reported in figure 1. Although it is not reported, it must be pointed out that, while SPI is a completely physical protocol, I^2C is typically developed with a more complicated software structure; this aspect can be seen as an advantage or an obstacle depending on designers preferences during the development of an application [4] [5].



	# PIN	OUTPUT STAGE	ADDRESSING	SPEED	DUPLEX
SPI	3+n	Push-pull	Chip Select	Limited by the devices	Full
I^2C	2	Open drain	In packet	Limited by RC constant	Half

Figure 1. Comparison between I^2C and SPI communication protocols.

The possibility of conjugating the benefits of these two protocols would lead to a communication standard particularly suited for smart objects and, therefore, for IoT applications. A fixed number of used pins would simplify the development of sub-systems, making easier to increase capabilities of a smart object; furthermore, by assigning to every device a long address, it would be possible to avoid the use of multiplexers, decreasing the circuital complexity. However, this can be done only if a very high speed is supported in order not to invalidate performances during a communication session; the protocol should also have the capability of recognizing new devices connected to the bus and eventually leasing shorter addresses to known smart objects. To unlock these features, it is clear that neither SPI nor I^2C, by themselves, can be used for these purposes.

These considerations led to the proposal and definition of *FlexSPI*: built on top of a classic SPI bus, it allows to completely share all its lines among different devices by leaving the slaves in high impedance until right before a communication session [1]. By means of a proper addressing form and the possibilities, for slaves, to signal the need of a communication session through the application of a pull-up resistor by the master, it is possible to obtain an high speed four wired protocol whose power consumption is dependent only on the devices themselves. Moreover, these features give the possibility of creating a series of procedures thought to obtain a self-sustaining bus particularly suited for smart objects, thanks to the possibility of safely hot-plugging sub-systems. Since this protocol is thought as a MAC layer over the legacy SPI bus, its implementation requires a further knowledge of advanced software architectures and how to combine them with the available modules of a device. For this reason, before studying the details of this communication protocol, the features of the hardware platform chosen for the implementation, a MSP430-family microcontroller, will be discussed, together with realtime operating systems and, in particularly, FreeRTOS [6]. This study aims to analyze the performances of both hardware and software, thus optimizing the firmware implementing FlexSPI and its shared communication bus.

II. UTILIZED HARDWARE AND SOFTWARE TOOLS

Before describing how FlexSPI works, the components used for its implementation are analyzed. The first one is the MSP-EXP430F5438 Experimenter Board, used to validate the written firmware; the second one, on the other hand, is an advance software architecture used to optimize performances called "real time operating system".

a. MSP-EXP430F5438 experimenter board

The system chosen to test the implementation of a FlexSPI firmware is the MSPEXP430F5438 Experimenter Board, shown in figure 2a. This evaluation board is made by Texas Instruments and is equipped with a MSP430F5438A microcontroller whose capabilities can be easily exploited thanks to the several peripherals connected [7]. The integrated interfaces on the board are a 138x110 dot-matrix LCD, a two-axis analog accelerometer, a 5-directional joystick, two push buttons, two LEDs and a full analog signal chain from microphone to audio output jack; it is also possible to exploit an UART communication thanks to the mini-USB connection. At last, wireless communication is also possible via TI wireless evaluation module headers or EZ430-RF2500T headers. With respect to photo in figure 2a, the position of these peripherals on board are shown in figure 2b [8] [9].

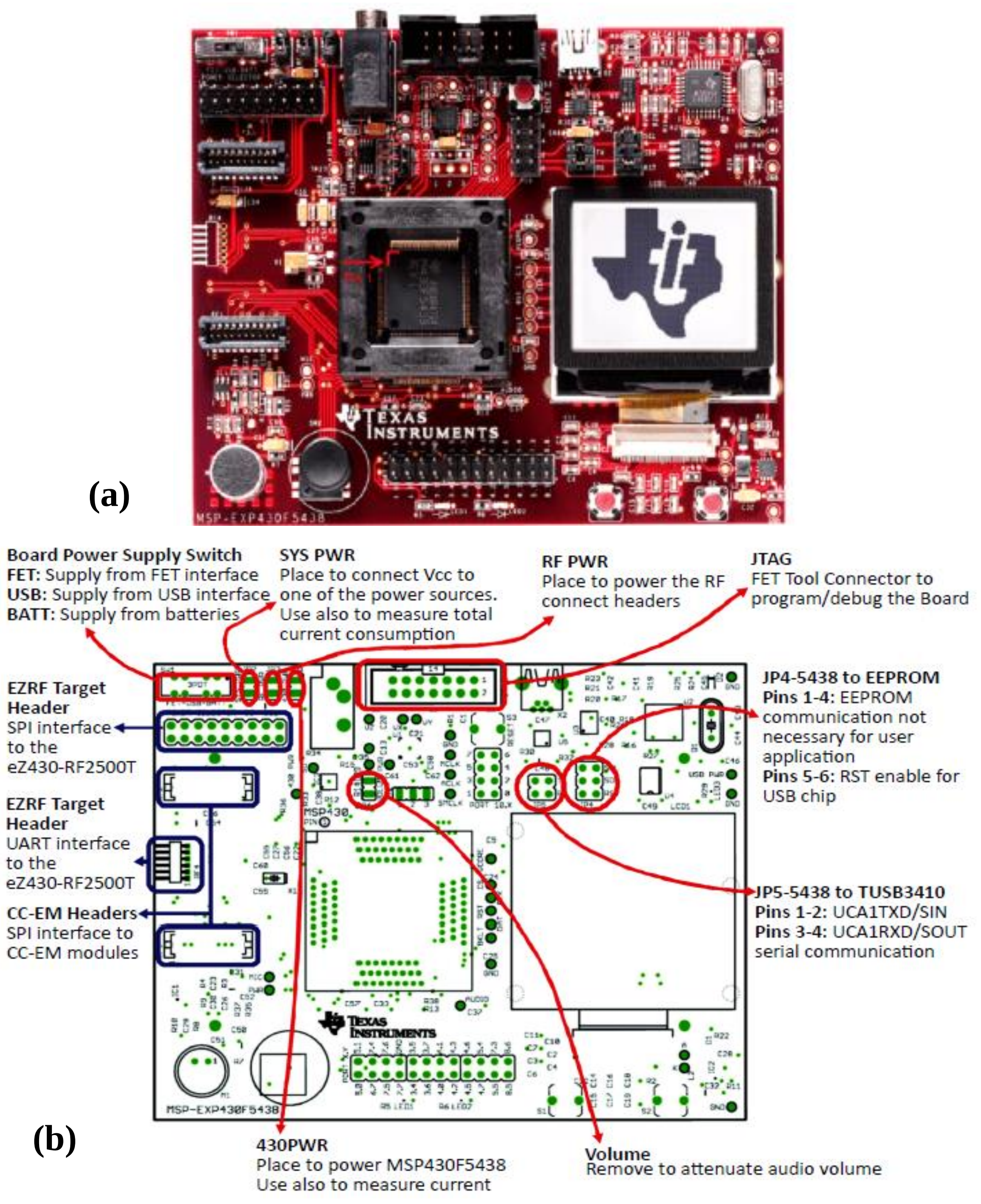

(a)
(b)
Board Power Supply Switch
FET: Supply from FET interface
USB: Supply from USB interface
BATT: Supply from batteries
SYS PWR
Place to connect Vcc to one of the power sources. Use also to measure total current consumption
RF PWR
Place to power the RF connect headers
JTAG
FET Tool Connector to program/debug the Board
EZRF Target Header
SPI interface to the eZ430-RF2500T
EZRF Target Header
UART interface to the eZ430-RF2500T
CC-EM Headers
SPI interface to CC-EM modules
JP4-5438 to EEPROM
Pins 1-4: EEPROM communication not necessary for user application
Pins 5-6: RST enable for USB chip
JP5-5438 to TUSB3410
Pins 1-2: UCA1TXD/SIN
Pins 3-4: UCA1RXD/SOUT serial communication
430PWR
Place to power MSP430F5438
Use also to measure current
Volume
Remove to attenuate audio volume
MSP-EXP430F5438

Figure 2. View of MSP-EXP430F5438 Experimenter Board (a) and its functional overview.

The microcontroller TI MSP430F5438A is very well suited for applications requiring thorough energy management thanks to its architecture that, combined with several low-power modes, makes possible to achieve an extended battery life [7] [8] [10]. Typical applications for this device include analog and digital sensor systems, digital motor control, remote controls, thermostats, digital timers, and hand-held meters [11] [12]. Among the features of this microcontroller, it is worth mentioning the powerful 16-bit RISC architecture of the CPU that, combined with constant generators, contribute to maximize code efficiency with a system clock up to 25 MHz. The CPU is highly transparent to the application: all operations, other than program-flow instructions, are performed as they were register operations, thanks to seven addressing modes for source operand and four addressing modes for destination operand [13] [14]. This architecture is reported in figure 3.

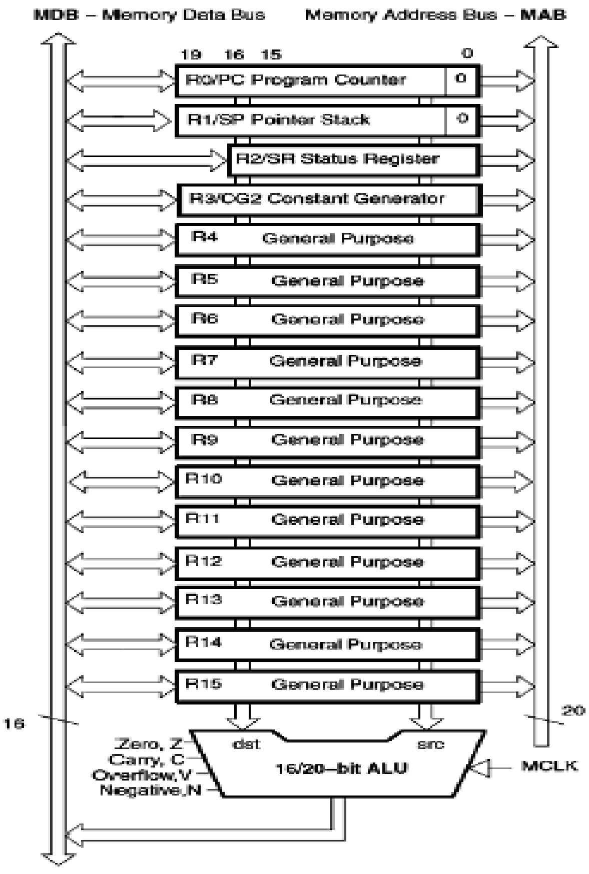


Figure 3. The MSP430F5438A CPU block diagram.

The CPU is equipped with 16 registers that provide reduced instruction execution time: the register-to-register operation execution time, in fact, is one cycle of CPU clock. Four registers, R0 to R3, are dedicated as program counter, stack pointer, status register and constant generator, respectively; remaining registers are general purpose registers. The different peripherals of the microcontroller are connected to the CPU using several kind of buses and can be handled with all instructions, which operate on word and byte data.

Another important feature is the digitally controlled oscillator, capable of waking up the device from one of the low-power modes to active mode in about 3.5μs. The MSP430F5438A has one active mode and six software selectable low-power modes of operation: the software can be written so that an interrupt event can wake up the device from any of the low-power modes, perform the required instructions and restore back to the low-power mode.

Regarding its storage capability, the MSP430F5438A is equipped with a 256 KB flash memory and a 16 KB RAM; memory is organized to store, among other things, the code memory and the location of the interrupt vectors. The CPU can perform single-byte, single-word and long-word writes to the flash memory while, regarding the RAM memory, it can power down different sectors to save leakage. At last, this microcontroller includes three 16-bit timers, a high-performance 12-bit ADC, up to four universal serial communication interfaces (USCIs: SPI, I2C, UART and their enhanced versions), a hardware multiplier, the DMA, an RTC module with alarm capabilities and up to 87 I/O pins. The complete block diagram of the MSP430F5438A is shown in figure 4.

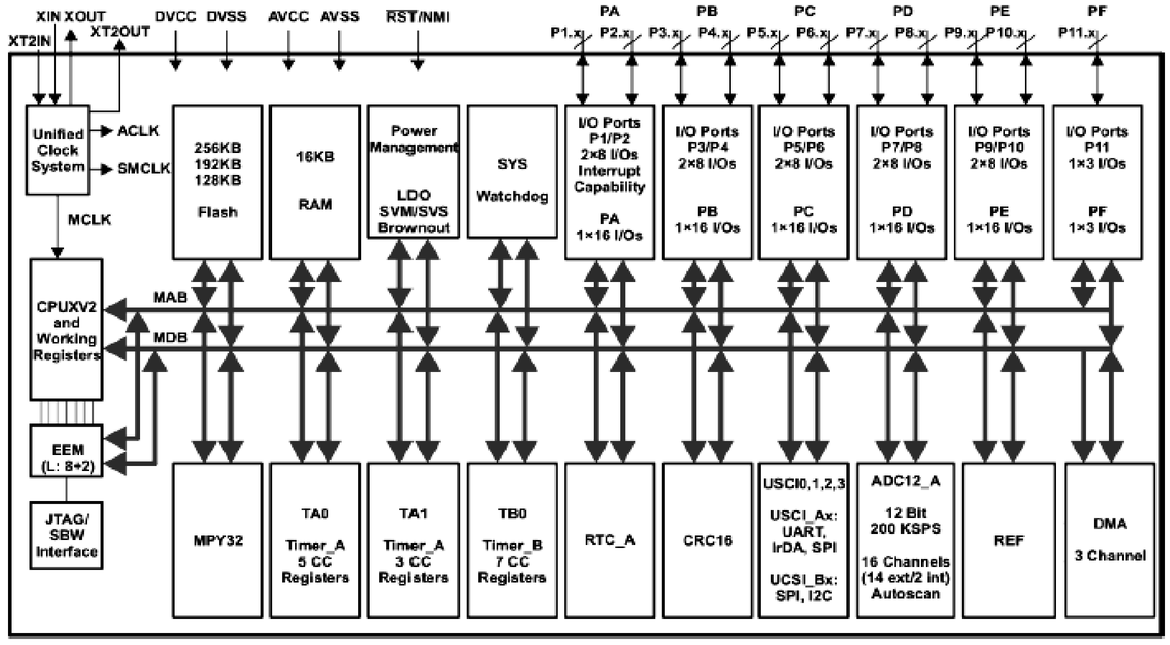

XIN XOUT
XT2IN
XT2OUT
DVCC DVSS
AVCC AVSS
RST/NMI
PA
PB
PC
PD
PE
PF
P1.x P2.x P3.x P4.x P5.x P6.x P7.x P8.x P9.x P10.x P11.x
Unified Clock System
ACLK
SMCLK
MCLK
256KB 192KB 128KB Flash
16KB RAM
Power Management LDO SVM/SVS Brownout
SYS Watchdog
I/O Ports P1/P2 2×8 I/Os Interrupt Capability PA 1×16 I/Os
I/O Ports P3/P4 2×8 I/Os PB 1×16 I/Os
I/O Ports P5/P6 2×8 I/Os PC 1×16 I/Os
I/O Ports P7/P8 2×8 I/Os PD 1×16 I/Os
I/O Ports P9/P10 2×8 I/Os PE 1×16 I/Os
I/O Ports P11 1×3 I/Os PF 1×3 I/Os
CPUXV2 and Working Registers
MAB
MDB
EEM (L: 8+2)
JTAG/ SBW Interface
MPY32
TA0 Timer_A 5 CC Registers
TA1 Timer_A 3 CC Registers
TB0 Timer_B 7 CC Registers
RTC_A
CRC16
USCI0,1,2,3 USCI_Ax: UART, IrDA, SPI USCI_Bx: SPI, I2C
ADC12_A 12 Bit 200 KSPS 16 Channels (14 ext/2 int) Autoscan
REF
DMA 3 Channel

Figure 4. Functional block diagram of the MSP430F5438A microcontroller.

b. Overview of the software architectures for embedded systems

One of the principles FlexSPI is based on is the abstraction from the physical layer in order to obtain not only a shared SPI communication but also many optional procedures; this approach, however, can only be fulfilled with a proper software architecture for the designed firmware. Any implemented firmware should be able to quickly respond to external events, even if it is in the middle of some processing; in some applications, a simple software architecture will be good enough, while, in some other cases, a more articulate approach should be preferred. A proper analysis can therefore be conducted only after a detailed study on interrupts and the problems that they generate.

An interrupt is a signal from the hardware regarding an event that needs immediate assistance from the microprocessor. Typically, as seen for MSP430F5438A, microcontrollers have dedicated pins, called *interrupt request pins*, that can be configured as interrupts sources, once connected to other devices. When the microprocessor detects that the interrupt pin is asserted, it stops data processing or other operations execution, saves its current status and performs the so-called *interrupt service routine* (ISR). An ISR, also called interrupt handler, is a portion of code with the instructions that must be performed as soon as an interrupt is requested; before its execution, the microprocessor saves on the stack the address of the instruction that would have been normally executed next. The interrupt routines should always be short and quick to let the device continue its previous processing, restoring the address of the next instruction from the stack. The classic situation that involves an interrupt in an embedded system firmware is described in the example in figure 5.

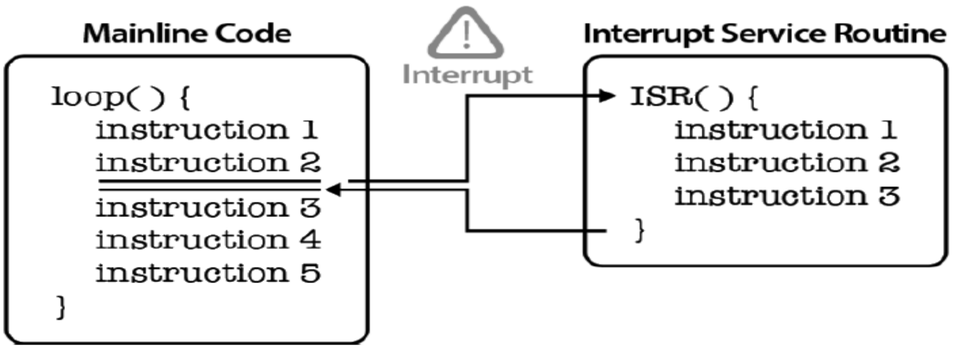
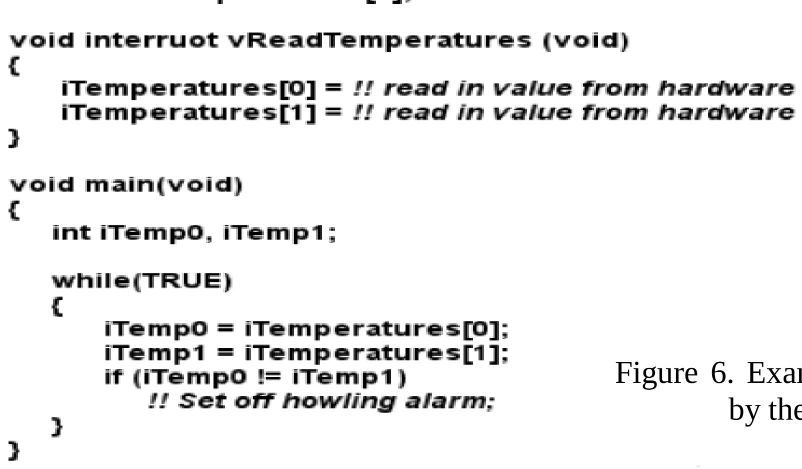


Figure 5. Example of main processing interrupted, causing the execution of an ISR.

One of the major problems when working with interrupts is that ISRs may modify a variable used by the main code, leading to the so-called *shared-data problem*. This annoying bug can be encountered when, for example, a situation like the one represented in figure 2.6 is faced: a temperature from two sensors is asynchronously measured and acquired with an ISR, and the values are compared in the main code. If the value changes between the copy of the measured values in the main code, the two "*iTemp*" variables will have a different value, although this is not true according to the measure, leading to a false alarm.

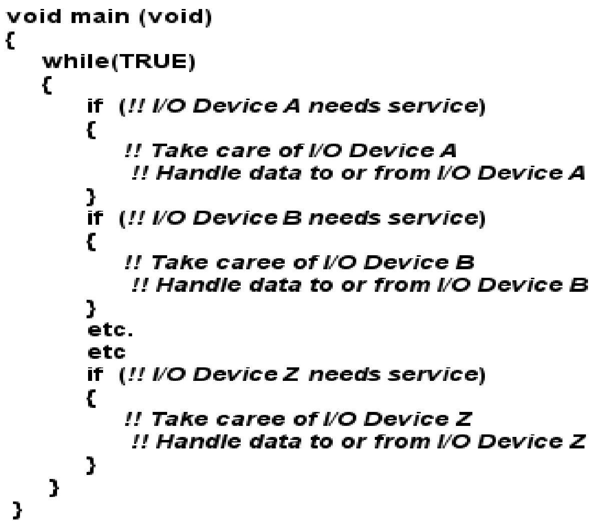


```
static int iTemperatures[2];

void interruot vReadTemperatures (void)
{
    iTemperatures[0] = !! read in value from hardware
    iTemperatures[1] = !! read in value from hardware
}

void main(void)
{
   int iTemp0, iTemp1;

   while(TRUE)
   {
       iTemp0 = iTemperatures[0];
       iTemp1 = iTemperatures[1];
       if (iTemp0 != iTemp1)
           !! Set off howling alarm;
   }
}
```

Figure 6. Example of code affected by the shared-data bug.

It should be pointed out that this kind of bug seldom shows up, but this makes it difficult to locate and can seriously compromise the integrity of written firmware. One could solve this bug by disabling interrupts before copying values and re-enabling them after; this solution is based on the observation that implicated portion of code is *atomic*, i.e. it cannot be interrupted. Although this approach solves problem, it brings some disadvantages related with interrupt latency. The interrupt latency is the time a system needs to respond to an interrupt, and it should be as low as possible; as said before, writing short interrupt routines can help. Latency greatly increases when disabling interrupts because, by doing that, the system can handle an interrupt (event that in any case will be registered) only when interrupts become active again, greatly slowing the system response time and therefore decreasing the performances of the embedded system. The shared-data bug, the interrupt latency and other similar aspects related to the firmware in embedded systems cannot be discussed without having a precise idea of the application one is building: in some cases, we can just ignore these problems since they will not affect system performances. This is the reason why several software architectures have been developed, with the idea of giving more control over the system response when needed.

The simplest architecture possible is called *round-robin*, prototyped with the example code in figure 7. This architecture, characterized by the total lack of interrupts, simply check periodically every device connected to the microcontroller, handling the read data and eventually performing other operations; an example of firmware based on this architecture can be the one implemented in a digital multimeter. This architecture results suited in applications when timing is not very important and the number of I/O devices connected is small. Although its simplicity can be enough for some systems, it remains very inadequate for responsive embedded systems, especially in an IoT perspective, characterized by devices whose response should be as fast as possible since they constantly interact [15].



```
void main (void)
{
   while(TRUE)
   {
       if  (!! I/O Device A needs service)
       {
           !! Take care of I/O Device A
            !! Handle data to or from I/O Device A
       }
       if  (!! I/O Device B needs service)
       {
           !! Take caree of I/O Device B
            !! Handle data to or from I/O Device B
       }
       etc.
       etc
       if  (!! I/O Device Z needs service)
       {
           !! Take caree of I/O Device Z
            !! Handle data to or from I/O Device Z
       }
    }
 }
```

Figure 7. Prototype of round-robin software architecture.

A more sophisticated architecture is the *round-robin with interrupts*. This architecture is similar to the previous one, with the important difference that interrupts can be exploited to signal urgent need of the microprocessor-based assistance. The possibility of giving a different priority to certain instructions improves firmware performance since also interrupts themselves can be handled with different priorities; this feature is restated in figure 8, where the two architectures are compared.

Once interrupts have been implemented, however, all the related problems discussed earlier arise, causing the need of much more scrupulousness when designing the code and, eventually, the accidental generation of bugs difficult to be tracked. The last software architecture proposed is the one that uses real-time operating systems, sometimes called *kernels*. As in the previous cases, interrupts have the highest priority and they signal to the main code that an action is required; the big difference is that this signaling is handled by the operating system itself that, moreover, decides which task needs to be executed next.

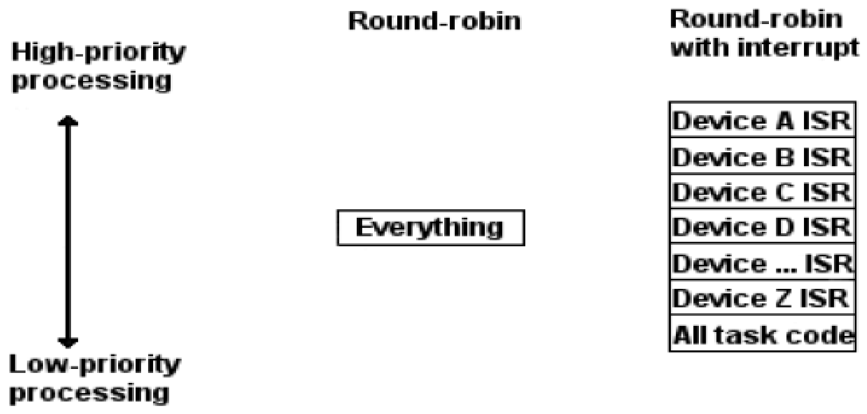


Figure 8. Priority levels for round-robin architectures.

As can be seen in figure 9, in fact, the so-called *main code* is itself divided in several tasks, with an assigned priority according to the designed application. In this way, also regular operations are subordinated to priority mechanisms.

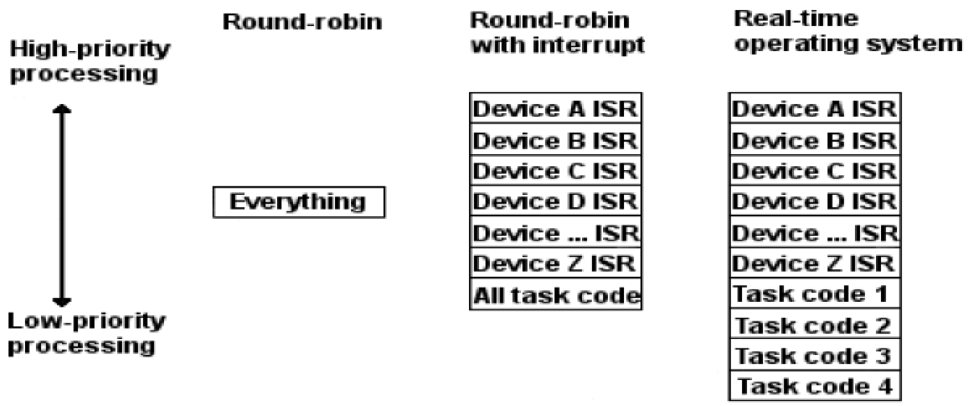


Figure 9. Priority levels for all the discussed architectures.

A task is simply a subroutine that, when needed, performs the required operations. In RTOSs the execution of a task is related to its belonging to one of this possible states:

- ✓ **Running**, when the microprocessor is executing the instruction belonging to that task;
- ✓ **Ready**, when another task is executing but, as soon as the microprocessor becomes available, it can be executed;
- ✓ **Blocked**, when the task has no pending operations and it is waiting an external event.

The different states a task can have are summarized in figure 10, where the transitions explain what causes the change of state. A task can have further states, according to the principles that developers choose to create their operating systems, but the listed ones belong to every implementation.

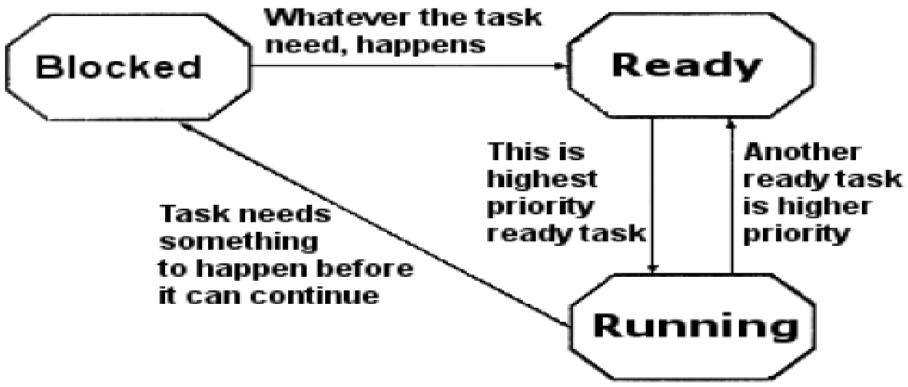

Blocked
Whatever the task need, happens
Ready
This is highest priority ready task
Another ready task is higher priority
Task needs something to happen before it can continue
Running

Figure 10. Typical task states in RTOS.

The most important component of a RTOS is the scheduler, responsible of keeping track of the state of each task and deciding which one should be executing: it constantly monitors the ready tasks, running the one with the highest priority and registering if a task has been unblocked or blocks itself. A task in fact, apart from being blocked if a higher priority task becomes ready, can block itself if it has no more actions to execute, waiting for further events for completing or restarting its operations. According to the implementation of the scheduler, an RTOS can be classified as:

- ✓ **Preemptive**, when the scheduler immediately blocks the running task if a higher priority one has woken up.
- ✓ **Non-preemptive**, when the scheduler waits for the running task to complete its operations first.

Every task, as shown in figure 11, possesses its own private context that includes register values, a program counter and a stack; all the other data, such as global or static variables, are shared among all tasks. This configuration, however, leads to the discussed problem of shared-data, since the scheduler could block a task using a variable in favor of another task

accessing the same data, causing a possible memory corruption. A first way to solve this problem is to write functions that are *reentrant*, i.e. that can be safely accessed by more tasks exploiting variables in a non-atomic way; however, sometimes this aspect is related to chosen compiler, therefore solution could not be universal. RTOSs, however, provide a powerful tool to efficiently deal with shared-data problem and to optimize designed firmware: *semaphores*.
A semaphore is a tool that ensures a secure management of shared-data among different tasks: before modifying a shared variable, the task can be forced to first attempt to take the semaphore, waiting if it is not available. As soon as another task releases the semaphore, the invoking task will successively take it, thus becoming able to safely modify the data; if other tasks want to modify the same variable, their attempt to take the semaphore will fail and they will have to wait, granting to the task that possesses the semaphore the chance to safely complete its operations. When this task completes its work, it will release the semaphore giving to other tasks the opportunity to modify the variable.

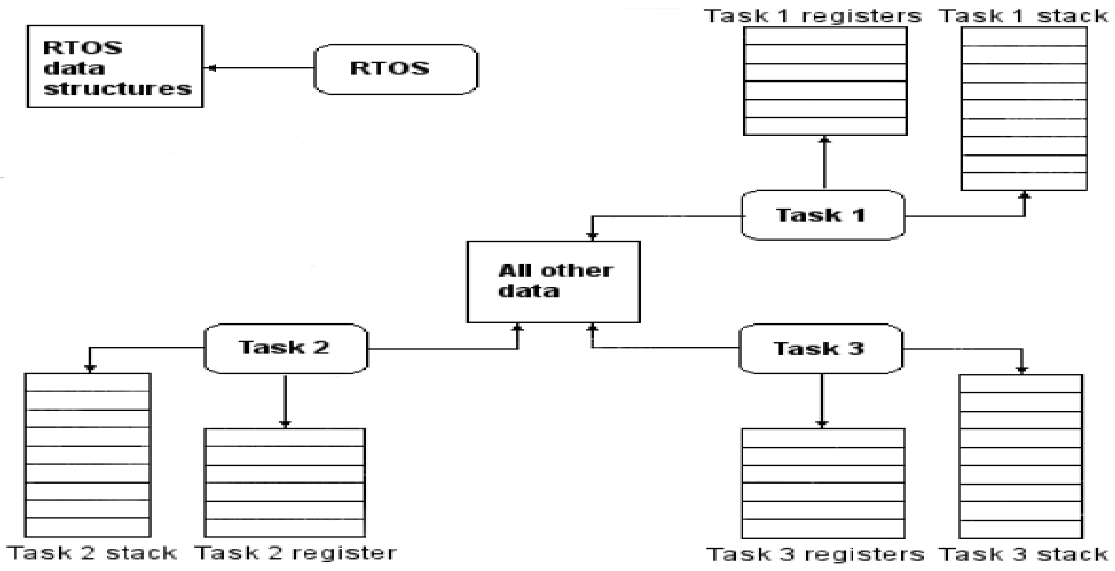


Figure 11. Typical data allocation in a RTOS.

There are several different kind of semaphores in RTOSs, used not only for dealing with the shared-data problems but also to optimize the timing of the whole firmware. These tools, together with many others, make possible to write a firmware composed by independently written tasks whose execution is autonomously regulated by the scheduler. This approach is particularly useful when abstraction from a pure physical world, e.g. pin toggling, is necessary, since one can safely design its code focusing on the different aspects of a particular layer in its stack, having to worry only that the interactions among layers are coherent. Abstraction also eases code management and extension, since modules are weakly linked to each others, granting to the developers the chance to add functions and procedures.

The offered tools and the exposed advantages give the opportunity to precisely tailor the written firmware to its application, leading to the consideration that RTOSs should be the primary choice when writing a firmware for any *IoT* application. Since the implementation of the FlexSPI firmware has been made working with a open source real-time operating system, exactly *FreeRTOS*, its main features will be now discussed.

c. The used Kernel: FreeRTOS

FreeRTOS is a real-time kernel on top of which embedded applications can be built to meet their hardware real-time requirements. Since the MSP430F5438A, as many microcontrollers, possesses only one core, only a single task can be in the running state: the scheduler decides which thread should be executed by examining the assigned priorities. It is a common trend to assign higher priorities to tasks that implement hard real-time requirements, and lower priorities to tasks that implement soft real-time instructions. This choice ensures that hard real-time threads are always executed before soft real-time threads, but priority assignment decisions are always more articulated and strongly linked to the application.

The building blocks of *FreeRTOS*, like all kernels, are tasks, implemented as C functions and not allowed to return parameters. Any task must be explicitly created in the main portion of the code through a given API that specifies, among other parameters, its priority. Once the scheduler is started, the task with the higher priority will be executed and, when its processing ends, a tick interrupt is processed, letting the scheduler decide which task should be executed next; an example of this situation is given in figure 12.

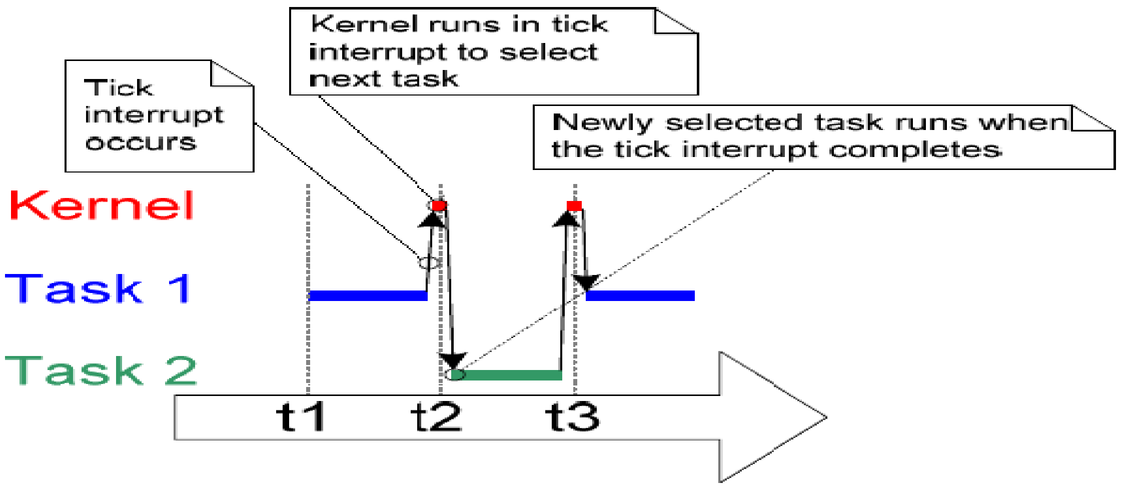

Kernel runs in tick interrupt to select next task
Tick interrupt occurs
Newly selected task runs when the tick interrupt completes
Kernel
Task 1
Task 2
t1
t2
t3

Figure 12. Example timing diagram describing the tick interrupt.

FreeRTOS supports all the classic task states plus another one called "*suspended*", i.e. when a task has been blocked and not available to the scheduler until it is properly resumed. Declared a proper handler, any task can suspend and resume itself or other tasks; it is also possible to change a task priority during the execution of the firmware. When no thread has to been executed, an "idle" task with the lowest priority and no instructions is running. FreeRTOS, however, gives the opportunity to exploit this idle task, obtaining the scenario in figure 13 to perform some minor actions or to put microcontroller in low-power mode. As soon as a task becomes ready, the scheduler quickly switches the execution thanks to preemption of idle task.

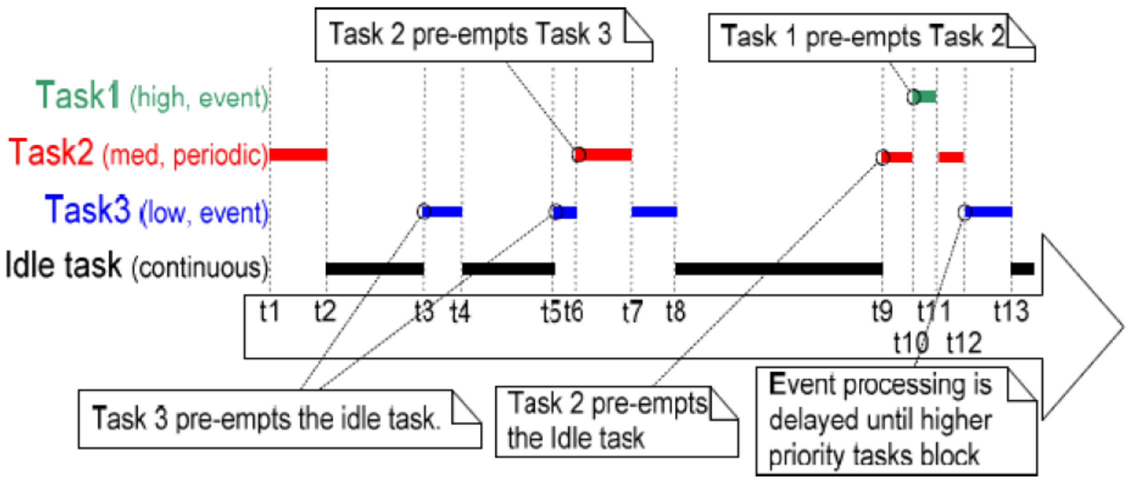

Task 2 pre-empts Task 3
Task 1 pre-empts Task 2
Task1 (high, event)
Task2 (med, periodic)
Task3 (low, event)
Idle task (continuous)
t1 t2 t3 t4 t5t6 t7 t8 t9 t11 t13
t10 t12
Task 3 pre-empts the idle task.
Task 2 pre-empts the Idle task
Event processing is delayed until higher priority tasks block

Figure 13. Example timing diagram with the idle task.

Interrupts are handled with a great variety of semaphores: typically, an ISR is synchronized with a task called "*interrupt handler*" by giving it the semaphore, as can be seen in figure 14. Thanks to this approach, it is possible to obtain really short interrupt routines and the remaining desired processing can be comfortably done outside.

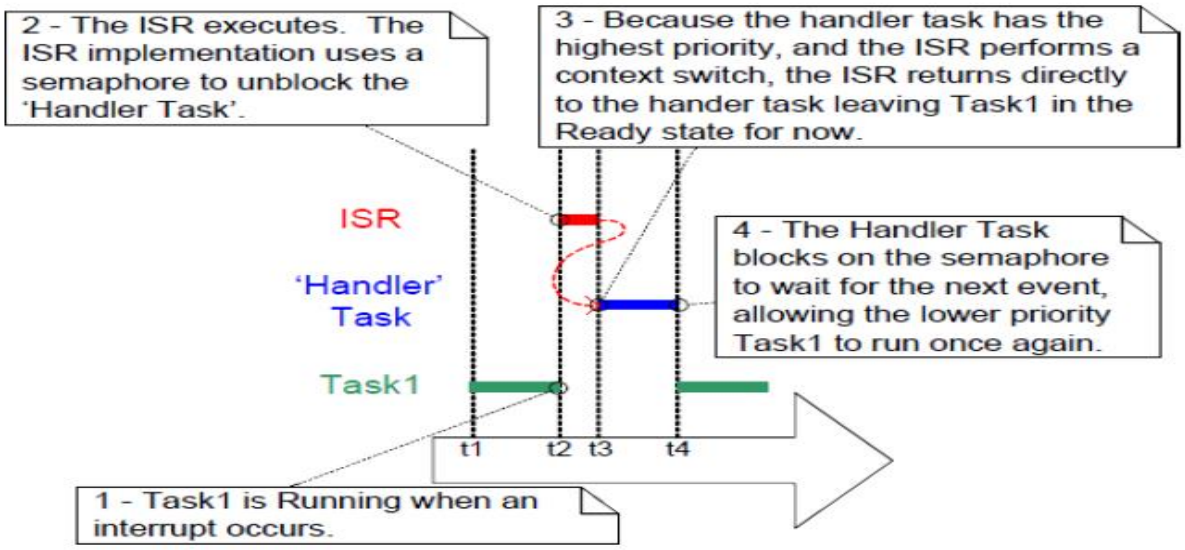

2 - The ISR executes. The ISR implementation uses a semaphore to unblock the 'Handler Task'.
3 - Because the handler task has the highest priority, and the ISR performs a context switch, the ISR returns directly to the hander task leaving Task1 in the Ready state for now.
ISR
'Handler' Task
Task1
4 - The Handler Task blocks on the semaphore to wait for the next event, allowing the lower priority Task1 to run once again.
t1 t2 t3 t4
1 - Task1 is Running when an interrupt occurs.

Figure 14. Timing diagram with synchronization between ISR and handler.

Semaphores can also be used to synchronize different tasks or to perform other functions; for this reason, apart from binary ones, in FreeRTOS there are:

- ✓ **Counting semaphores**, semaphores that can be given and taken more than once, giving the opportunity to latch multiple interrupt events.
- ✓ **Mutexes**, special type of binary semaphores used to control access to a shared resource, avoiding the rising of the shared-data problem with a mutual exclusion scenario.

FreeRTOS semaphores are, actually, a special type of another instrument called *"queues"*. A queue is a *First Input First Output* (FIFO) buffer used by tasks to exchange data with each others; in this sense, a binary semaphore is simply a one-item long queue that transfers a token granting the possibility of execution. When a task writes a data in the queue, if not explicitly specified differently, it will be placed in the tail of the queue, while data are extracted from the head. If a task tries to write in a full queue or to read from an empty queue, it will block for a configurable amount of time, and then it will try to perform again the same action. An example on how tasks that use queues are executed is reported in figure 15.

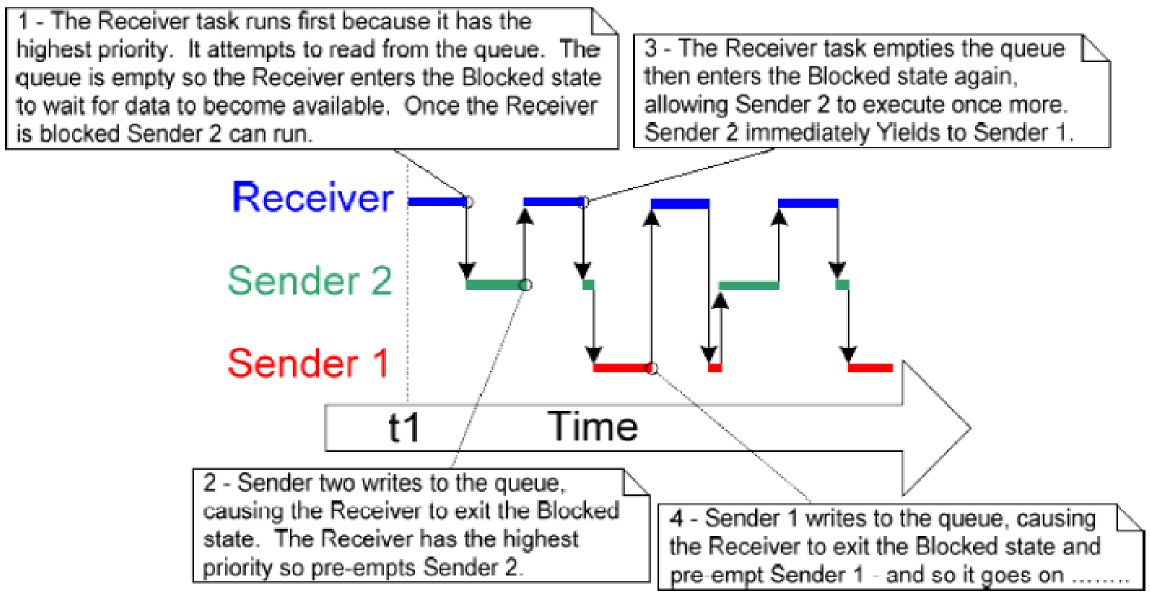

1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Once the Receiver is blocked Sender 2 can run.
3 - The Receiver task empties the queue then enters the Blocked state again, allowing Sender 2 to execute once more. Sender 2 immediately Yields to Sender 1.
Receiver
Sender 2
Sender 1
t1
Time
2 - Sender two writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.
4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ....

Figure 15. Example timing diagram of a data transfer through a queue.

Queues can be used to transfer data encapsulated in structures from multiple sources; in this case, some identification mechanism should be provided. However, if the size of the stored data is very large, it is preferable to use the queue to transfer a pointer to the data, leaving to the receiver task its recovery and processing. With this approach, it is possible to speed up the data exchange and save the available RAM.

By choosing this approach, it is preferable to dynamically allocate the necessary memory before passing the pointer; moreover, the code structure must assure that only one task will access the memory while allocating the memory, modifying its content and eventually freeing it. For this reason, FreeRTOS uses two ***API***, *pvPortMalloc* and *vPortFree*, that grant a safer approach with respect to the C classic *malloc* and *free* functions. In the developed FlexSPI firmware, among the different possibilities, the definition of these two functions declared in the *"Heap_4.c"* file has been chosen: the safety of memory allocation and release is granted by temporarily suspending the scheduler before the necessary operations, resuming it when processing ends. This description of FreeRTOS has been obtained not only through the analysis of its manual [6]. The frameworks used to characterize peripherals on the MSP-EXP430F5438 experimenter board have been once again written using this operating system. Although listings are not reported, the expressed considerations in description of FreeRTOS are also the results of experimental development of firmware based on this advanced software architecture.

III. DESCRIPTION OF THE FLEXSPI FUNDAMENTALS

The purpose that has led to the ideation of this communication protocol is obtaining a fully shared SPI bus, with a fixed amount of wires, without renouncing to the advantages of a push-pull output stage and obtaining an architecture capable of great flexibility [16]. All the four signals of a classic SPI protocol are entirely shared by the slaves on the bus: when a master wants to communicate with a particular device, it will perform an addressing at packet level, whose specifications will be discussed later. All slaves will download the transmission and check if they are the addressed receivers: if so, they will continue to process the packet, otherwise it will be discarded. Thanks to this approach, it is possible to provide not only unicast addressing but also multicast and broadcast messaging.

It is possible to obtain this behavior by redefining *Chip Select* role in the communication session: instead of just bytes, this signal is used to window messaging sessions. This line is, in fact, used to wake up the slaves and let them know that an incoming transmission is about to begin. Further transmissions will be handled by slaves according to some built-in finite state machines, updated according to the content of the previous packet. This approach shows no problems when applied to the SIMO signal but can provoke a serious number of dangerous conflicts when referred to SOMI line, if an arbitration rule is not applied; moreover, slaves are allowed to use this line to asynchronously request the master attention. Potential malfunctions are prevented by configuring the SOMI pin, when not employed in a data transfer, as a high impedance input, while the master uses a pull-up resistor to mark the line in a recessive state. As soon as the master needs to receive data from a particular slave, it explicitly gives the possession of the SOMI line and disables the pull-up resistor, granting only to that device the right to occupy the channel with its transmission. An example of a possible bus using FlexSPI is shown in the following figure 16 [17].

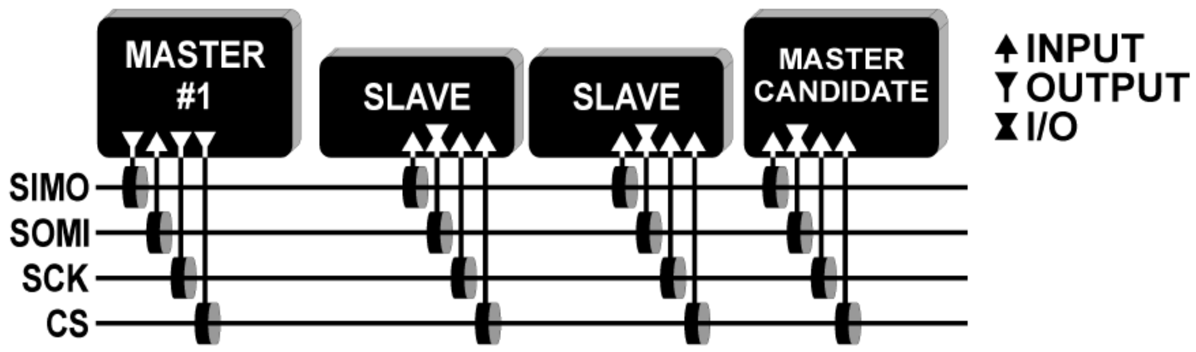


Figure 16. Example of FlexSPI bus.

This picture shows also one of the possible consequences of employing this bus: on the same lines, the mastership of the channel can be negotiated through a proper packet exchange and, if requested by application, a slave can become the master. It is also possible, theoretically, to create partitions of slave devices on a FlexSPI bus by having different chip select lines, providing an useful tool to include on the bus SPI-only devices; their inclusion, although explicitly supported by this protocol, should be carefully implemented.

The exclusive concession of the channel from the master does not forbid, as stated before, any form of slave signaling: the presence of the pull-up resistor, in some circumstances, can be exploited to perform an open-drain signaling. This option has been forecast in applications where, for example, the master is simply used as a sink collecting data from slaves, e.g. sensor [18] [19]; with this configuration, any device has the possibility to communicate the need of a bus concession to the master, which will perform a procedure to understand which slave produced the signal. Once pinpointed the slave, the master will give to it possession of the channel, downloading the content of the slave's pending queue and eventually transmitting data. FlexSPI, in fact, is based on SPI protocol and therefore supports full-duplex communications, doubling the available bandwidth of channel with respect to I^2C; this eventuality, however, must be properly signaled in order to let devices correctly to configure their pins and available memory. It is obviously possible to have just half-duplex data exchanges but, if transmitter is the slave, the master should be made aware of how many clock pulses it must send to fully download the packet.

Analyzing this protocol from a more physical point of view, the principal feature of FlexSPI is taking advantage of both the GPIO and SPI modules available on microcontrollers: the default situation, in fact, features the SIMO and SOMI signals as inputs for both the master and the slaves, while the remaining signal are output for the master and input for the slaves. As soon as a particular communication session is desired, the firmware will disconnect the necessary pins from the GPIO module and connect them to the SPI one, performing the required operations. This model of the physical interface is represented in figure 17, showing that the constant multiplexing among modules is completely transparent to application layer.

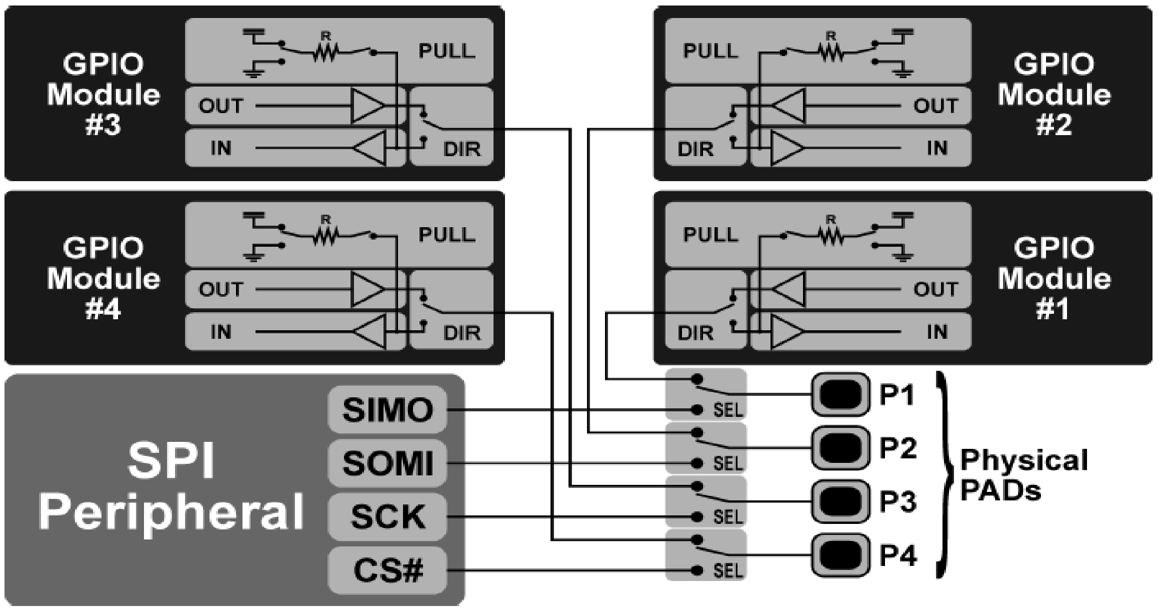


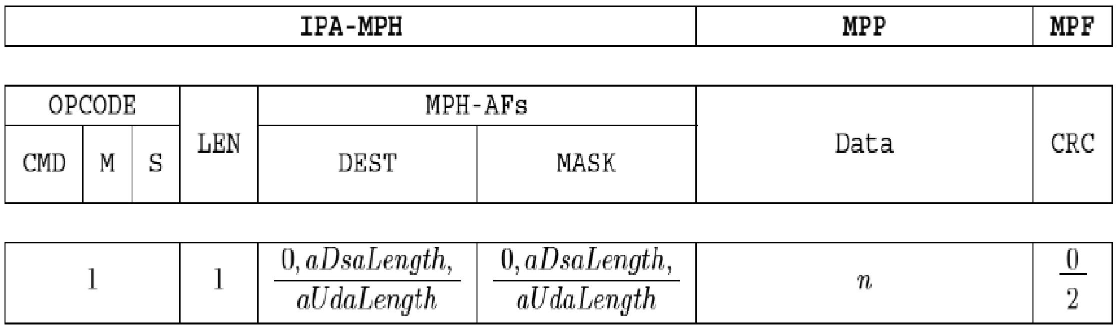
Figure 17. FlexSPI reference physical layer.

An important implementation advice, adopted in the developed firmware, has been followed: it is preferable, for the master, to never connect the *Chip Select* line to the SPI module: since this signal is used to window command session and not bytes, in fact, its behavior would be more unpredictable. It is therefore suggested to let *Chip Select* pin always connected to the GPIO module, lowering and rising its output logical level when needed in the implementation.

a. Packet structures and addressing strategies

FlexSPI uses two different kinds of address categories: *Universal Device Address* (UDA) and *Device Short Address* (DSA). *UDAs* are several bytes long universal addresses that unambiguously represent every single device that can be connected to a FlexSPI bus; these addresses are fixed and cannot be altered during the entire device lifetime. Only two addresses are reserved: all zeroes, used if an *UDA* is not assigned and all ones, for broadcast sessions. *DSAs*, instead, are much shorter than *UDAs* and therefore used to reduce protocol overhead; these addresses are leased by the master that controls the channel by mean of a dedicated procedure. Even in this case the addresses with all zeroes and all ones are reserved with analogous functions. Two types of addressing strategies can be used, *In-Packet Addressing*

(IPA) and *Off-Packet Addressing* (OPA), whose features will be analyzed later; for a full understanding of frame formats, it is sufficient to say that while the first case performs addressing at packet level, the second one exploits a different strategy. Since addressing is performed only towards slaves, an asymmetric packet format was created to reduce overhead and power consumption, especially in slaves; this is the reason why only masters can have two different frame formats. If IPA is used, the master packet will have the structure in figure 18, where the name of the various fields and their length expressed in byte is reported.



IPA-MPH						MPP	MPF
OPCODE			LEN	MPH-AFs		Data	CRC
CMD	M	S		DEST	MASK		
1			1	$\frac{0, aDsaLength,}{aUdaLength}$	$\frac{0, aDsaLength,}{aUdaLength}$	n	$\frac{0}{2}$

Figure 18. IPA Master packet format.

The packet structure is composed by different fields and sub-fields whose role is here reported:

- ✓ **IPA-Master Packet Header** (IPA-MPH): header of the packet, only the first two fields are mandatory. It is composed by:
  - **Opcode**: specifies the packet content or the command; it is composed by:
    - ➢ **CMD**: marker of the command.
    - ➢ **M**: indicates if the packet contains a MASK field.
    - ➢ **S**: if set, the address is a DSA, otherwise it is an UDA.
  - **LEN**: represents the residual length of the packet, i.e. the number of bytes following the LEN field itself.
  - **Addressing Field** (MPH-AF): is the field used to address one or multiple slaves; in some procedures may be omitted. This field is composed by:
    - ➢ **DEST**: is the address of the receiver and it must be always set.
    - ➢ **MASK**: used to send unicast, multicast or broadcast packet.
- ✓ **Master Packet Payload** (MPP): is the informative content that the master wants to send to one or more slaves; it can be empty if the master is sending commands.
- ✓ **Master Packet Footer** (MPF): optional field that can be used to make a safer communication by using a CRC signature and check.

Since in OPA, addressing is not performed at packet level, there is no need of addressing fields in the packet header (figure 19, with reduced header). The only different field from IPA format is *ELEN*, a two-bit long field that indicates a total packet length greater than 257 bytes.

OPA-MPH			MPP	MPF
CMD	ELEN $2b$	LEN	Data	CRC
1		1	n	$\frac{0}{2}$

Figure 19. OPA Master packet format.

Slaves are not allowed to perform addressing and therefore their packet structure is the one in figure 20. The asymmetry in the frame structure has been chosen because, in sophisticated systems, slaves are typically energy-constrained devices that would benefit from a shorter transmission. All fields have the same meaning of an IPA master packet, with the only difference in the two last bits of the *OPCODE*:

- ✓ The penultimate bit is reserved and always 0.
- ✓ P, instead of S, used to signal the presence of pending data in the slave's queue that should be sent to the master in another session.

SPH				SPP	SPF
LEN	OPCODE			Data	CRC
	CMD	0	P		
1	1			n	$\frac{0}{2}$

Figure 20. Slave packet format.

Once discussed packet formats, the two different addressing techniques must be analyzed. The first one is *In-Packet addressing*, that uses frames, as shown in figure 18, to perform a packet level addressing. A device will consider itself the receiver of a packet if the following equation holds true: $M[i] \odot (D[i] \oplus S[i]) = 0 \, \forall i$

where: *M[i]*, *D[i]* and *S[i]* are the i-th bit of, respectively, the mask field, destination address field and the slave self address; $\oplus$ B is the XOR logical operator, $\odot$ is the AND logical operator. The mask field is therefore used to let more devices being addressed from the same frame coming from the master, providing the support for multicast and broadcast messaging.

Off Packet Addressing can be exploited only by devices with a set DSA. This technique is used to avoid the need of SOMI reservation through IPA and can be very useful when multiple communication sessions are required to minimize the overhead, performing a lighter privileged data exchange with a device. In order to avoid conflicts among devices, however, developers must ensure that every connected device support this addressing technique. The procedure that realizes OPA is called *Chip Select Prioritization (CSP)*. This technique is performed with the signaling in figure 21 and it is articulated in three steps. The first one is the initialization, highlighted in light gray in the following figure: it is a falling edge of the clock with the chip select high, i.e. when no communications are happening at the moment. This event is detected by all the slaves on the bus that will perform the necessary operation to get ready for the very beginning of the *CSP* procedure.

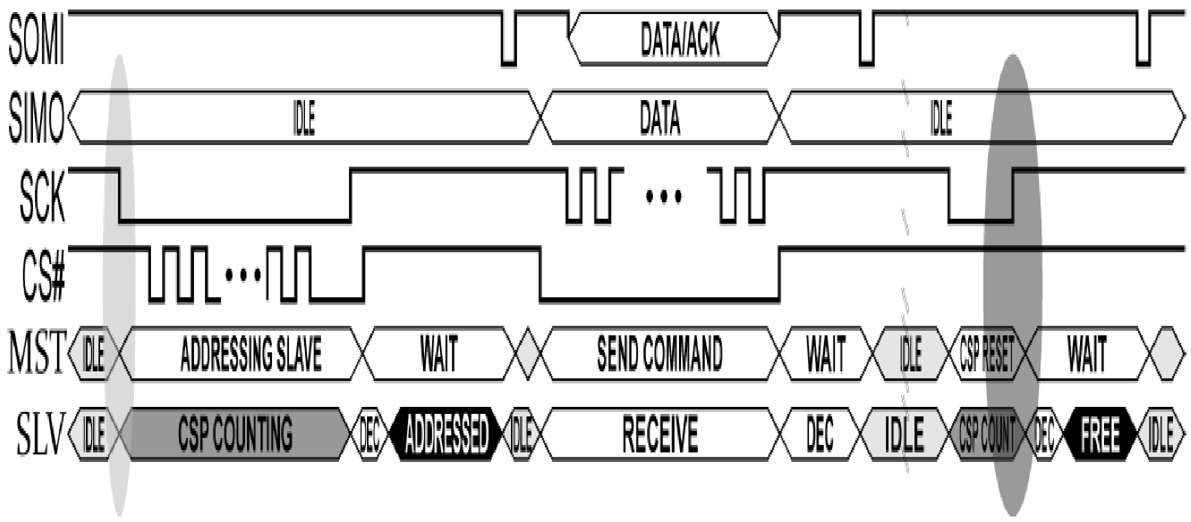


Figure 21. CSP signaling of the master (MST) and the addressed slave (SLV).

Addressing is performed by toggling the *Chip Select* line for a number of times equal to the addressed DSA slave. This count goes on until the clock returns high, signaling to slaves that they can stop counting the chip select pulses. The device that finds the count equal to its address will consider itself the receiver, while the others will discard further communications. The addressed slave will take possession of the SOMI and start a communication session according to what has been required from the application layer of the devices; the possession is held until the reset sequence is received. Highlighted in dark gray, the reset signal is a clock falling edge followed immediately by a rising edge, always with the *Chip Select* high; this event is an invalid zero count of chips select pulses that announces, to all the slaves on the bus, that channel reservation is ended. The master can therefore address a new slave with the same procedure knowing that all the devices are ready to listen again to the channel.

b. Data Transfer Mechanisms

FlexSPI main features and advanced procedures are enabled by sharing the SPI bus between slave devices, but this technique needs a mechanism to avoid collisions to work properly. This is the reason why the *SOMI* line is always kept free by all slaves on the bus while the master marks it with a recessive state by applying a pull-up resistor and, moreover, giving to slaves open-drain signaling capabilities. This signaling is performed in two different ways thanks to many procedures that will be analyzed in the following paragraph. The concession of the channel is given by the master with a *poll-slave* packet: this command is used to inform the addressed slave that in the following communication session it will have to send data.

Both full-duplex and half-duplex messaging are allowed in SPI peripherals, and so in FlexSPI, doubling the channel available bandwidth. It must be considered, however, that since master is responsible for injecting clock pulses into the bus, it must be made aware of the amount of necessary toggling to download the slave packet. Therefore, if communication is half-duplex and the slave is device that must send data, the master will first download just the first byte of a slave packet, containing the residual length of frame and then perform necessary operations to send as many clock pulses as required. In practical implementations, this behavior can be obtained by configuring the master to send dummy data: it is a slave's job to ignore the incoming transmission. This dependency on frame length must be carefully taken into account if the transmission is full-duplex: a possible data loss can be experienced if devices are not ready to react to different packet lengths. Presence of the *LEN* field in packet headers help devices to properly identify who needs to send a longer frame and promptly react:

- ✓ If the slave is sending a longer packet, the master must keep sending bytes although its packet ended with the *CRC* bytes, padding the transmission with a count-down to 0 of the remaining clock cycles that will be sent.
- ✓ If the master, on the other hand, is supposed to send a longer packet, the slave will simply start sending padding bytes as long as clock cycles are received.

This behavior is summarized in figure 22, where the added bytes in both cases are at the receiver side after the CRC field, with the "PB" label used to identify slave's padded bytes. As soon a communication session is over, the master deselects the slave by rising the *Chip Selec*t. The SOMI is then freed by slave and the master applies the pull-up resistor, recovering the idle situation ready to listen to open-drain signals or to start a new session. Commands sent by the master must be interleaved by a proper delay to let slaves process the received packet; however, an ad hoc procedure can be employed to avoid the use of imprecise delays.

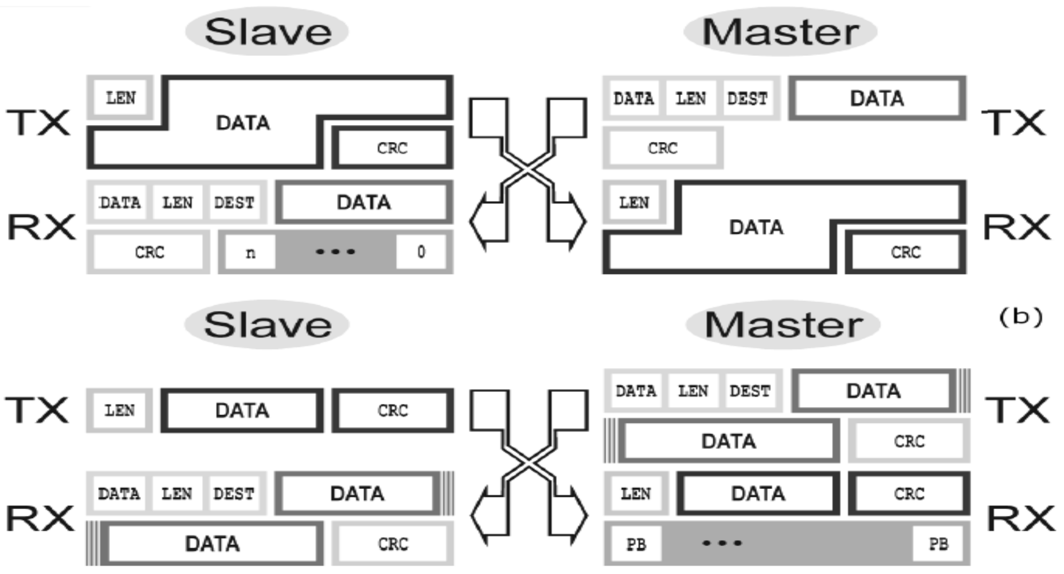


Figure 22. Packet exchange in a full-duplex communication if slave LEN field is greater (a) or master LEN field is greater (b).

IV. OVERVIEW ON THE AVAILABLE PROCEDURES

Once described mechanisms involved in the creation of a shared SPI bus and, consequently, the fundamentals of a FlexSPI-based data exchange, some of the available procedures are here described. Thanks to its features, in fact, FlexSPI is capable of providing a series of advanced tools to help monitoring the bus condition, to ensure an efficient and fully supported communication by all devices and, above all, to create a dynamic master-slave architecture with the possibility of safely hot-plugging new devices.

a. MAster Device Somi based IRQ Synchronization (MADIS)

The *MAster Device Somi based IRQ Synchronization (MADSIS)* procedure has been designed to create a synchronization mechanism between the master and slave at the end of a communication session in order to avoid imprecise delays between transmission of different packets. When developing the application layer of the firmware, it must be taken into account the eventuality that some devices on the bus may not support this sub-procedure; on the other hand, the master possesses not only the address of the slave on bus but also their FlexSPI capabilities. Once a transmission between the master and a slave is successfully ended, instead of waiting for a defined amount of time, the master can be configured to wait a signal on the SOMI line, already marked with a pull-up resistor. The slave, on the other hand, as soon as it finishes to process the received packet, is allowed to produce a short open-drain signal on its output line. In this way, the master will be informed that the last addressed slave is ready to start a new communication session. The master device waiting time cannot be

infinite: it is a developer's job to size a proper wait time for a SOMI event according to the designed application, in order not to excessively slow down the bus. This waiting wait, however, does not excessively affect the master energy consumption since the device can be put in a low-power mode with interrupts enabled, ready to react to the expected signal.
If the transmission from the master was addressed to more than one slave, the open-drain signaling grants that all slaves can produce a SOMI pulse without any kind of conflict. However, since slaves may have different CPU capabilities, in case of multicast transmissions, it is recommended to let slaves wait a random delay before pulling down the SOMI line: in this way, slower devices can safely complete their processing avoiding conflicts and data loss.

b. SIMO/SOMI Link Indicator

The *SOMI* signal is typically disconnected from the SPI module and used as a GPIO pin for both master and slave, with the master responsible for applying a pull-up resistor to provide open-drain signaling capabilities to slaves. This situation, however, can be exploited by slaves that, by monitoring the logical level of SOMI line, can use the provided piece of information as indicator of link's health: this is what the *SOMI-Link Indicator (SOMI-LI)* procedure does.

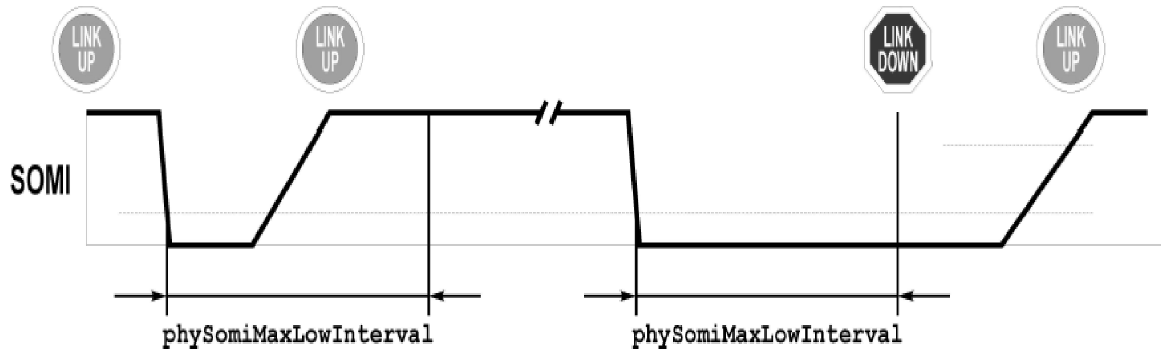


Figure 23. SOMI-LI timing diagram.

As can be observed in figure 23, when the SOMI logical level is high, slaves deduce that this link is valid. As soon a transition is detected, every slave will start a timer: if the SOMI line does not return high in a certain amount of time, the link will be considered broken. A similar technique can be applied to *SIMO line* when *Chip Select* is inactive, enabling a monitoring procedure for slaves called *SIMO Link indicator (SIMO-LI)*. In this procedure, as shown in figure 24, a slave produces a short pulse on *SIMO line*; the master has already applied a pull-up resistor with monitoring purposes. Similarly to the SOMI-LI procedure, if its logical value returns high in a defined amount of time, the slave will deduce that a valid master is currently managing channel. On the other hand, if timeout expires, the link will be considered broken; this event can be seen as a warning related to excessive fanout or a master serious malfunction.

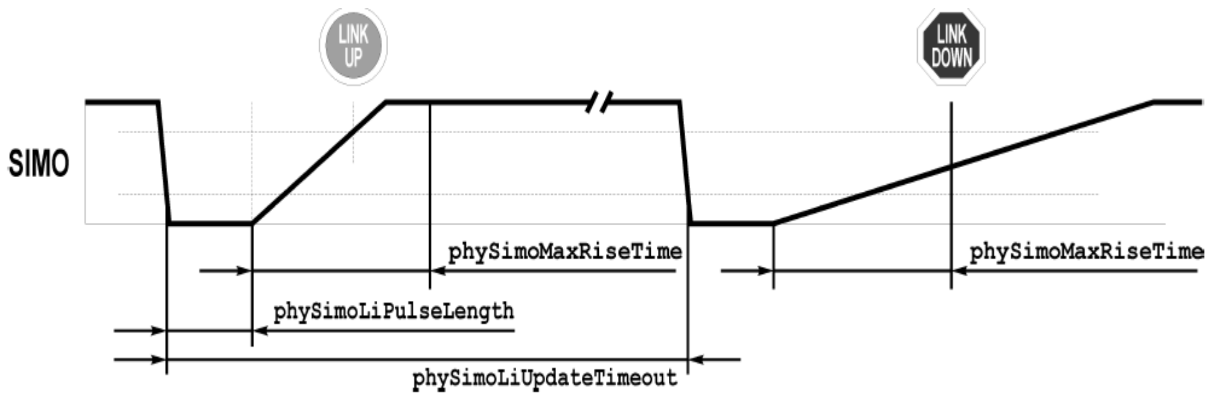


Figure 24. SIMO-LI timing diagram.

c. PHY Speed Negotiation

An IoT application can be characterized by deeply different devices that are supposed to communicate with each other; in order to minimize data loss, a suitable speed must be chosen every time by the main entity of a smart object, to communicate with other sub-systems. For this reason, FlexSPI supports an optional procedure called *PHY Speed Negotiation (PSN)*, whose purpose is to let to slaves negotiate with the master a suitable clock speed for their data exchanges. This procedure employs two commands: *SPEEDREQ* and *SPEEDACK* (as shown in figure 25). A slave, if the channel is available, uses a communication session to send a *SPEEDREQ* packet having, as payload, the proposed speed. The master, then, processes the request and, if found acceptable, replies with a *SPEEDACK* packet having the accepted speed as payload, otherwise the payload will be a different speed proposed by the master itself. Regardless of the payload, once the *SPEEDACK* packet has been received by the slave, it will send a *SPEEDACK* packet too; this is done to ensure that the proposed speed has been recognized and granted by all devices. To ensure a safe data transfer, if the master supports different speeds for different slaves, any multicast communication session is performed transmitting data with the lowest speed among the ones negotiated.

d. Procedure of Master Device Solicitation

The nature of a classic SPI communication is based on a master that sends data but is also responsible to download them from slaves that are not able to signal the urge of master's assistance to empty their pending buffer. FlexSPI, based on a continuous commutation among the SPI and GPIO modules, grants to slave the possibility to signal to the master need of a communication session in order to transfer data. This feature can be implemented in two different ways, both exploiting open-drain connection of SOMI line when no data exchange is happening: *Slave Initiate Master Polling* and *SIMO enabled Slave IRQ Signaling*.

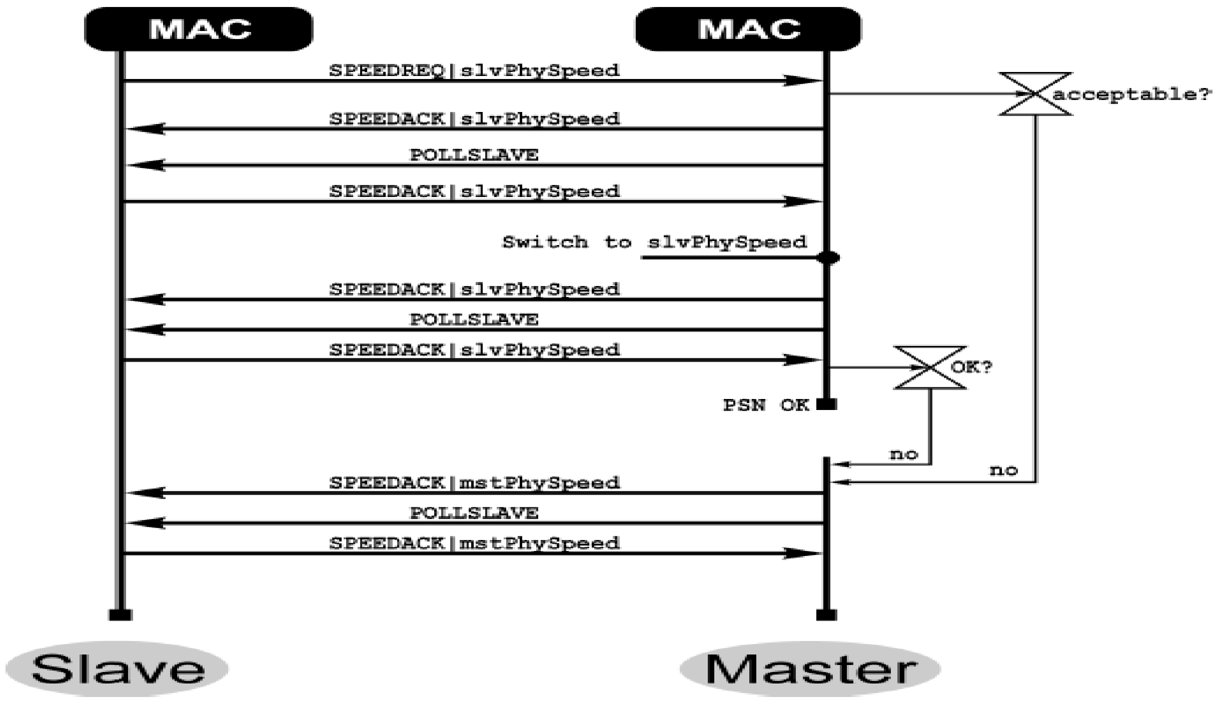


Figure 25. Packet exchange model for the speed negotiation procedure.

Slave Initiate Master Polling (SIMP) is an optional procedure initiated by a slave that needs to send data. The slave first checks the *Chip Select line* and, if found inactive, assumes that the channel is not occupied and therefore feels authorized to start the procedure. The slave, thanks to the open-drain connection, can send a short SOMI pulse to the master: this choice avoids collisions if more than one slave needs to send data. As soon as the event is detected by the master, as can be noted in figure 26, a session of slave polling begins.

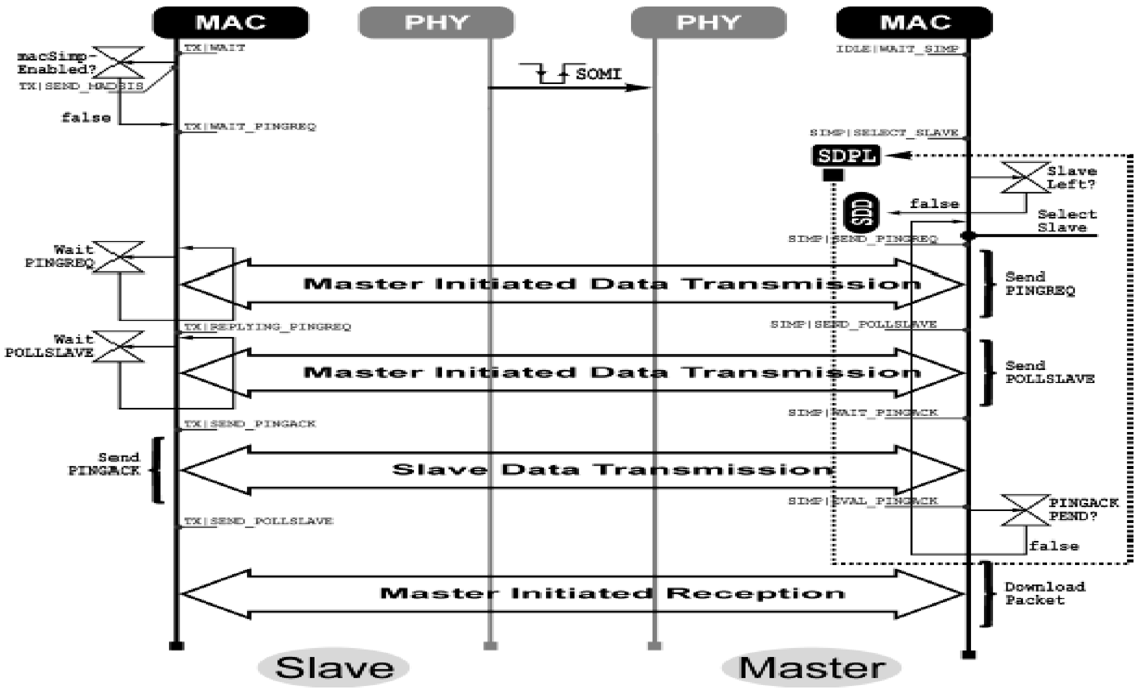


Figure 26. Communication sessions for SIMP procedure.

The master, in fact, cannot understand who produced the signaling and, therefore, has to send a *PINGREQ* command to every slave and a subsequent *POLLSAVE* command to concede the channel and let the addressed slave reply. The *PINGACK* command used by slaves to reply will have a ***P*** field set to 1 if the polled slave has pending data: as soon one of the replies to the master's ping has this feature, the master will recognize the source of the SOMI pulse and therefore starts the procedure to download pending data. If no slave replies with a *PINGACK* having a P field set, the master will deduce that a new device has been connected to the bus and, consequently, will start a proper procedure to identify its address.

It must be noted that, since SOMI pulses will not collide, more than one slave can safely start a *SIMP* procedure; however, only the first slave polled by the master and having a pending data will be served, leaving the others unattended. For this reason, if a slave finds its SIMP request unsatisfied, it is forced to start again the same procedure.

SIMO enabled Slave IRQ Signaling (S^2IS) is the alternative optional procedure thought to provide master solicitation while avoiding slaves' polling loop, as in the case of *SIMP*. This procedure is entered only if devices possess a DSA address. This method, as can be seen in figure 27, exploits both the *SOMI* and *SIMO* signals to notify the urge of a communication session while providing proper identificator of the slave starting procedure. The slave requiring assistance produces a short pulse on the *SIMO* line while starting as soon as possible to toggle the *SOMI* line, otherwise false *SIMO-LI* notifications could be generated. The identification of the slave comes from the *SOMI* line, toggled as many times as the slave's *DSA*.

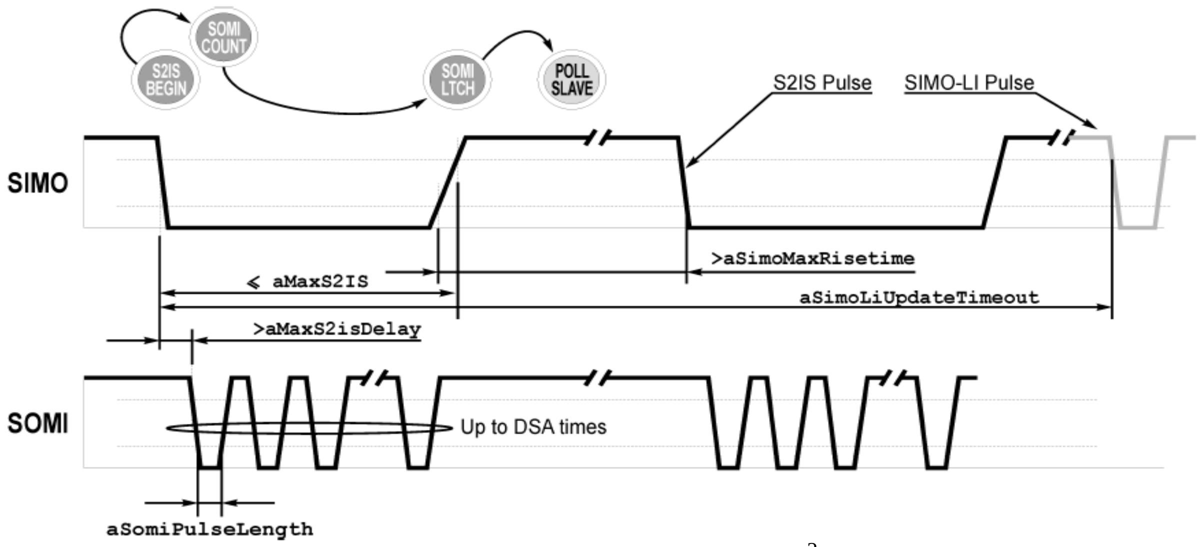


Figure 27. SIMO and SOMI lines during the S^2IS procedure.

Once the master has recognized the requesting slave, it promptly produces a *POLLSLAVE* command to reserve it channel and then downloads pending packet. As in previous procedure, if, for some reason, the master does not find the slave address in its memory, it will be forced to start a polling session to identify the source of message. When devices on bus don't support any of these two procedures, a third one can be used: called *Periodic Slave devices Polling*, it is aimed to request from time to time to slaves if they have pending data to be transmitted.

e. Slave Discovery Procedure

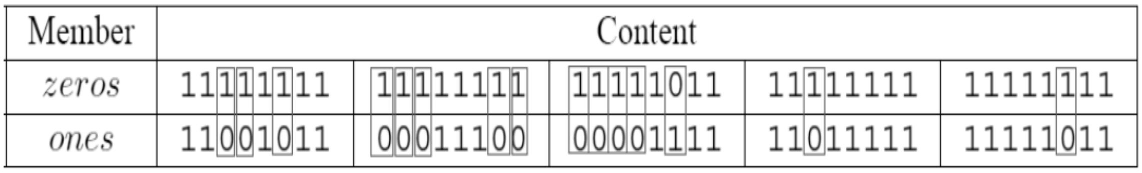
One of the most powerful features of FlexSPI is providing a mechanism to register presence of new devices on the bus, identifying their addresses and eventually their capabilities to optimize future communication sessions. The *Slave Discovery Procedure (SDP)* is responsible for the complex task of identifying new addresses and can be entered, for example, when the master is solicited by a slave to give the channel but it is not able to identify it, due to the lack of its identity in memory. The basic principle of this procedure is obtaining multiple responses to particular ping requests, exploiting the open-drain connection of devices. This mechanism can be safely used avoiding collision if slaves reply with a special *PINGACK* packet, crafted with all the header fields as zeroes when responding to a multicast ping. To speed-up the procedure, it is not necessary that slaves wait for a *POLLSLAVE* to reply: the master will promptly begin to inject clock pulses in order to retrieve any null frame.

Supposing more than one slave on bus, the first step to identify the address of a new device is writing an address conflict table: this analysis is used to investigate if slaves share the same value in some position of their addresses. This preliminary step requires, apart from *PINGREQ* packet, the *BCASTSHUT*: this command tells to addressed slaves not to reply to the following ping request. Among its options, the *BCASTSHUT* command can silence devices having particular address structures or addresses greater than a certain threshold.

The conflict table is filled with the following procedure:

1. A *BCASTSHUT* aimed to silence slaves whose addresses have only the LSB equal to 1 is sent, followed by a *PINGREQ* broadcast command.
2. If an answer is received, this means that at least a slave with a 0 as LSB in the address is present on the bus. The master will then set the column corresponding to the LSB, on a row called zeroes; if no answer is detected, the cell will be clear.
3. The same procedure is repeated, this time silencing slaves with a 0 as LSB. A reply will produce the setting of the LSB column in the ones line, otherwise a clear.
4. The procedure is repeated, shifting the investigated bit to the left until all the bits composing the address are investigated.

In this way, a table as the one in figure 28 is obtained. From its analysis, it is possible to deduce that where no conflict results, i.e. when in a column there is a 0 and a 1, all slaves share the same value of the bit in that position of their address.



Member	Content				
zeros	11111111	11111111	11111011	11111111	11111111
ones	11001011	00011100	00001111	11011111	11111011

Figure 28. Example of conflict table.

The number of conflicting bit positions can be calculated with the following formula:

$$\#(c) = \sum_{j=1}^{\#byteAdd} \sum_{i=1}^{8} zeros_{ij} \odot ones_{ij}$$

where $zeros_{ij}$ and $ones_{ij}$ are the i-th bit of the j-th byte of the respective row. Given this result, it is possible to conclude that the number of bits of the address that must be investigated are *8 x #byteAdd - #(c)*. This preliminary step allows to exponentially reduce the number of possible addresses that a slave can have, lightening the employed algorithm search by applying it only to the conflicting positions.

The shared bus architecture is exploited with a binary search applied in a distributed way: each slave device will actively participate in the discovery, since they are the only ones possessing the required pieces of information. This mechanism is called *distributed Binary Search Algorithm (dBSA)* and it uses once again *PINGREQ* and *BCASTSHUT* commands; this time, however, addresses greater than a certain value are silenced. A proper set of flag variables ensures that, when ping answers are reactivated, only slaves having an undiscovered address will consider themselves authorized to reply, ensuring that discovered slaves will not take part anymore in the process. In this way, the algorithm becomes faster the more it is able to find new devices. This cooperative discovery needs a proper mapping function to apply the binary search only to conflicting bits. The function is the following:

$$M_{CA}(\gamma) \longrightarrow \forall i \in [1, \#(a)]\ \alpha_i = \begin{cases} \gamma_j & \text{if } O_i = 1 \wedge Z_i = 1 \\ 1 & \text{if } O_i = 1 \wedge Z_i = 0 \\ 0 & \text{if } O_i = 0 \wedge Z_i = 1 \end{cases}$$

where Z_i and O_i are the i-th members of the *ones* and *zeros* rows, *#(a)* is the length in bits of addresses and *j ϵ [1; #(c)]* is an index that must be updated every time the first condition of the previous equation is applied. Thanks to this equation, it is possible to place investigated values of the binary search, i.e. bits in conflicting position, in a valid candidate address and to verify its belonging to bus. The resolution of conflict bits through *dBSA* to discover unknown addresses of new devices is made by the algorithm described with the flow-chart in figure 29. Two indexes are initialized, *HSide* and *LSide*, and their mean value, *Pivot*, is calculated; this value is mapped with the previous equation and then used by the *BCASTSHUT* and *PINGREQ* commands to find whether that address belongs to the bus. The value of *Pivot* is continuously updated with the classic approach of a binary search, until a new address is finally discovered. Once the new slave address is registered, nothing is known regarding its FlexSPI capabilities; the protocol however implements an *ad-hoc* command, *GETOPT*, to collect these pieces of information. The master can also decide to start procedure to assign a *DSA* address to lighten the overhead of future communications or directly start a communication session with it.

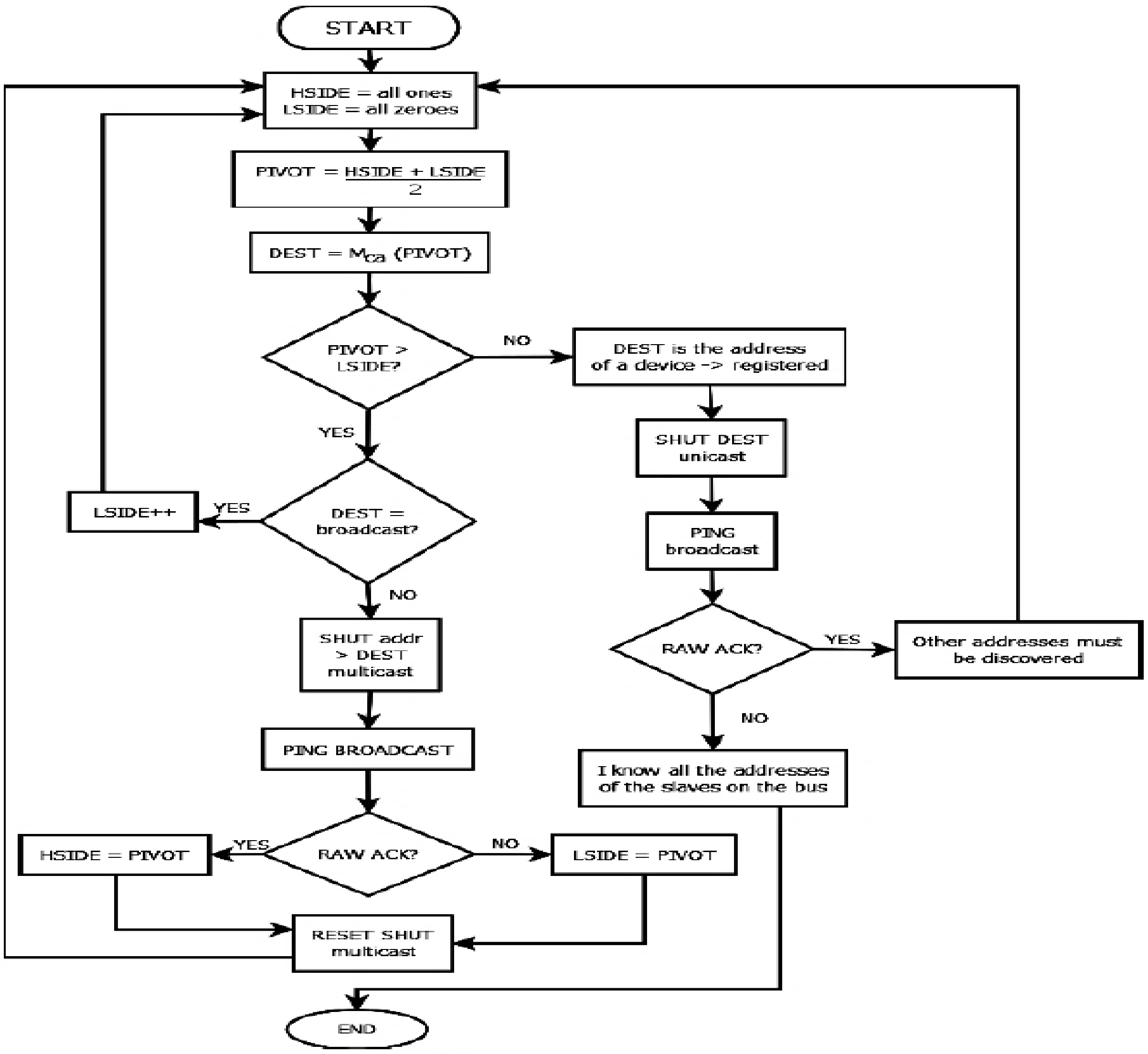


Figure 29. Algorithm flow chart related to conflicting bit resolution.

V. IMPLEMENTATION OF FLEXSPI AUXILIARY PROCEDURES

This paragraph is focused on the implementation and testing of some of the advanced procedures that FlexSPI enables; in particular three procedures, among those outlined in paragraph IV, have been selected. By adding these features to the developed framework, it is possible to explore and appreciate the expandability of this communication protocol, making it suitable to meet advanced requirements of smart objects. Some preliminary improvements have been applied to the firmware in order to increase its efficiency:

- ✓ *FreeRTOS* tick rate has been reduced from 1 kHz to 2Hz. This variable sets the frequency of the RTOS tick interrupt that, when occurs, wakes up the scheduler to investigate if a task has become ready; this situation is encountered, for example, when the device is executing the idle task waiting for a delay to end. Although this modification does not particularly affect procedures, it becomes very important in terms of energy consumption: in this way, the device remains in low power mode for the entire wait period.
- ✓ The implementation of short delays has been obtained by keeping the device active, while long delays exploit the on-board *Real-Time Clock*. This peripheral can be used to produce a time interval that relies on the hardware counter module equipped in the microcontroller. In this way, it is possible to wait for long delays without the CPU intervention, that can be operated, therefore, in low power mode.

The three chosen advanced procedures are: ***MADSIS***, ***SOMI-LI*** and ***PSN***, described with their implementations and with results coming from experimental validation tests.

a. MADSIS procedure: code design and validation

MAster Device Somi based IRQ Synchronization (MADSIS) procedure allows slaves to pull-low in an open-drain fashion the *SOMI line* as soon as they have concluded processing related to a communication session. This signal collected by master is an indication that device is ready to start a new session, providing a safe synchronization mechanism. This technique can be used if the master is enhanced with ability of recognizing interrupts coming from *SOMI pin*, feature obtained by tying SOMI pin to a GPIO one. The stack of layers is the following one.

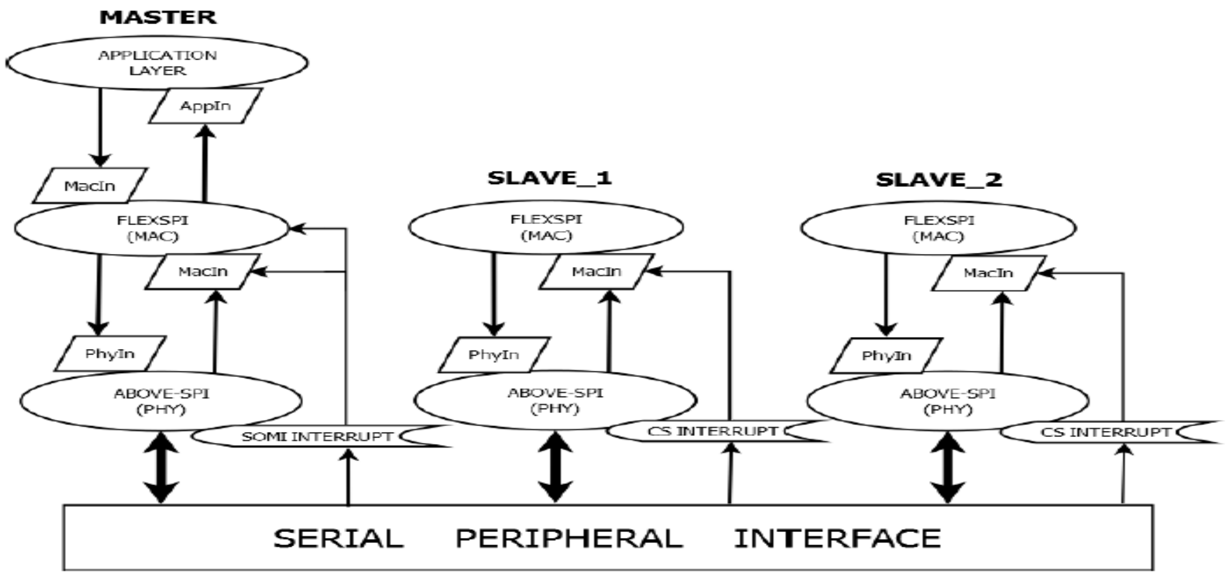


Figure 30. FlexSPI layered stack modified for MADSIS.

As can be noted, however, when a SOMI interrupt is detected by the master, the MAC layer is informed in two different ways. This behavior has been implemented with the listing reported below, where the *PORT1 Interrupt Service Routine* of the master is reported.

```
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR(void){
 portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
  switch(P1IV){
  case(0xE):
    P1IFG &= ~(1<<6); //event on SOMI
    if(macState != Unemployed){//received madsis
      xSemaphoreGiveFromISR(xSemMadsis ,&xHigherPriorityTaskWoken); //MADSIS
    }
    else{ //madsis
      xQueueSendFromISR(xQueueMacIn, (void *) &pVdataPlmeSOMIeventIndication , &
          xHigherPriorityTaskWoken);
    }
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    break;
  default: break;
  }
}
```

In order to speed-up the system, in fact, a SOMI interrupt is interpreted according to *macState*:

- ✓ If *unemployed*, the master has received a totally asynchronous event. For this reason, since the master is supposed to react by potentially performing complex actions, the event is communicated to the MAC layer via its listening queue.
- ✓ On the other hand, when the master is moving across its finite state machine, this event can be generated only by a slave at the end of a communication session. For this reason, it is used to wake up the master letting it to continue its operations.

The last situation is the one involved during the MADSIS procedure. After resuming the idle condition of its SPI pins, the MAC layer of the master receives a *PlmeTXdisableRequest* or *PlmeRXdisableRequest* primitive and it begins to wait for *SOMI* events from previously addressed slave. The master, in fact, attempts to take a *semaphore* called *xSemMadsis* and blocks itself. As soon as the SOMI event is generated by slave, ISR releases the semaphore and the MAC layer becomes running thus resuming its operations, eventually beginning a new data exchange. The *MADSIS* procedure requires also to modify the slave portion of the firmware: as soon as it completes its processing on the received packet, a SOMI event is generated by invoking a function called *madsisSlave*. Implementation of this simple function is reported in the next listing. SOMI pin, left floating by slave, is configured as output and a zero logical level is produced; after that, the pin returns to its previous condition. When this function is invoked, the pull-up resistor was already applied by master; for this reason other operations are not required.

```
void madsisSlave (void){
 extern pin_t SOMI;
 set_out(SOMI);
 outLevel_low(SOMI);
 set_in(SOMI);
}
```

MADSIS procedure requires also hardware-level operations on the master: interrupts on SOMI line can be obtained by short circuiting this pin with a GPIO one. In this way, it is possible to extend PORT1 interrupts to the SPI pins. The model of experimental setup used to verify the MADSIS procedure, together with a detailed photo showing the short circuit on the master, is represented in figure 31; red wire and clamps were used to perform connection on the master.

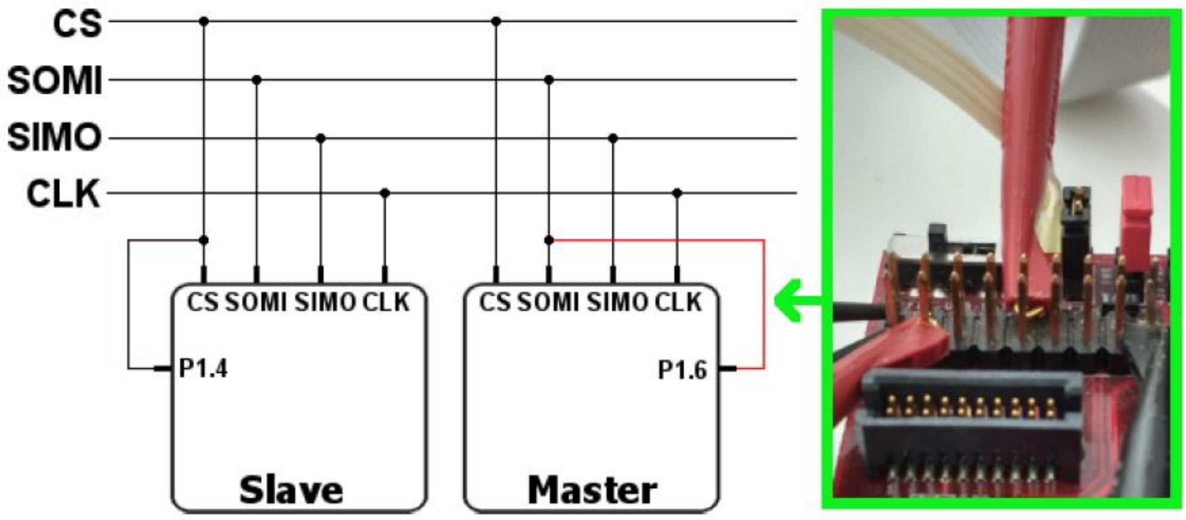


Figure 31. Model of the experimental setup for MADSIS with detail of required short circuit.

Once the procedure has been implemented both at hardware and software level, its verification, together with eventual observed benefits, was conducted by preparing discussed experimental setup; figure 32 shows all the devices with their short circuits, together with the logical analyzer. A single slave pinging procedure has been carried out and, thanks to provided support of MADSIS procedure, it has been possible to remove the delays that in the previous framework slowed down the communication. The bus speed has been set to 500 kHz.

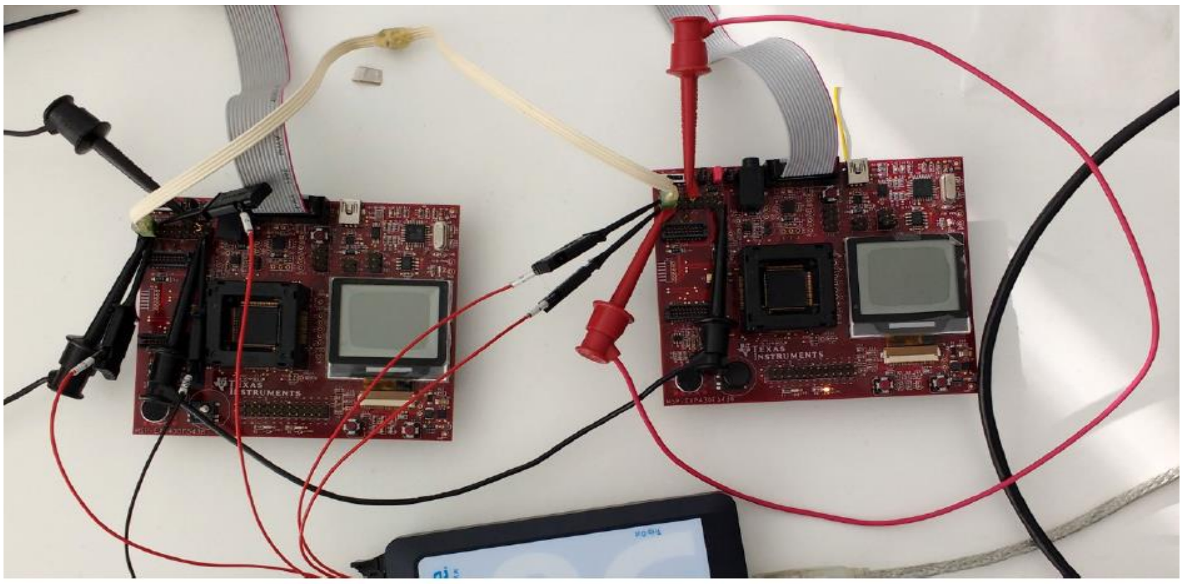


Figure 32. Realized experimental setup for the MADSIS procedure.

The good outcome of the procedure has been verified by collecting the exchanged signals on the bus with the logic analyzer. The resulting waveforms are represented in figure 33, where SOMI pulses have been highlighted and zoomed: according to logic analyzer, their duration is ~ 5µs; this is the time it takes for SOMI line to return high due to the pull-up resistor.

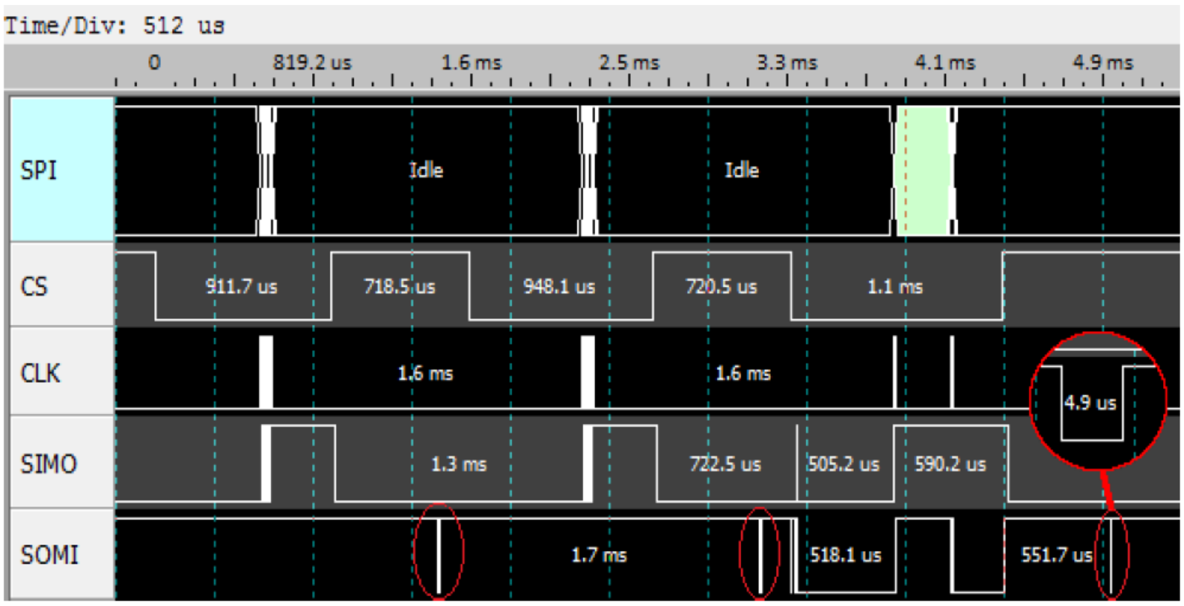


Figure 33. Signals during the MADSIS test; SOMI pulses are circled in red.

Thanks to this procedure, it is possible to reduce the inter-leaved time among two communication sessions; the overall time required to perform any action has been therefore lowered and the system speeded up. Moreover, slave devices having different computation capabilities can be used without having the need to wait for the worst-case.

b. SOMI-LI Procedure: Code Design and Validation

SOMI-Link Indicator (SOMI-LI) procedure is the technique that allows slaves to check bus condition by monitoring the SOMI pin: since in idle condition this pin is marked with a pull-up resistor by the master, if its logical level becomes suddenly low, slaves can investigate whether the link with the master has been compromised. When a broken link is detected, the device running FlexSPI should react according to what is decided by higher layers of the application, once informed. Given the experimental nature of this framework, it has been decided to use a task parallel to the MAC layer, *vSomiLiTask*, that verifies the bus condition: if the link results broken, the task itself performs some programmed operations. The block diagram with the stacked layers is therefore expanded as reported in figure 34.

This procedure was tested by disabling master application layer, monitoring bus during the absence of operations: slave SOMI interrupts are enabled only if *macState* is *Unemployed*. The *Interrupt Service Routine* associated to the slave *PORT1* has been expanded: SOMI pin in fact has been short circuited to a GPIO pin to monitor a sudden falling edge of the line. This modification is shown in listing below, where the ISR is reported; since device should remain ready in case a communication session is about to start, Chip Select interrupts remain active.

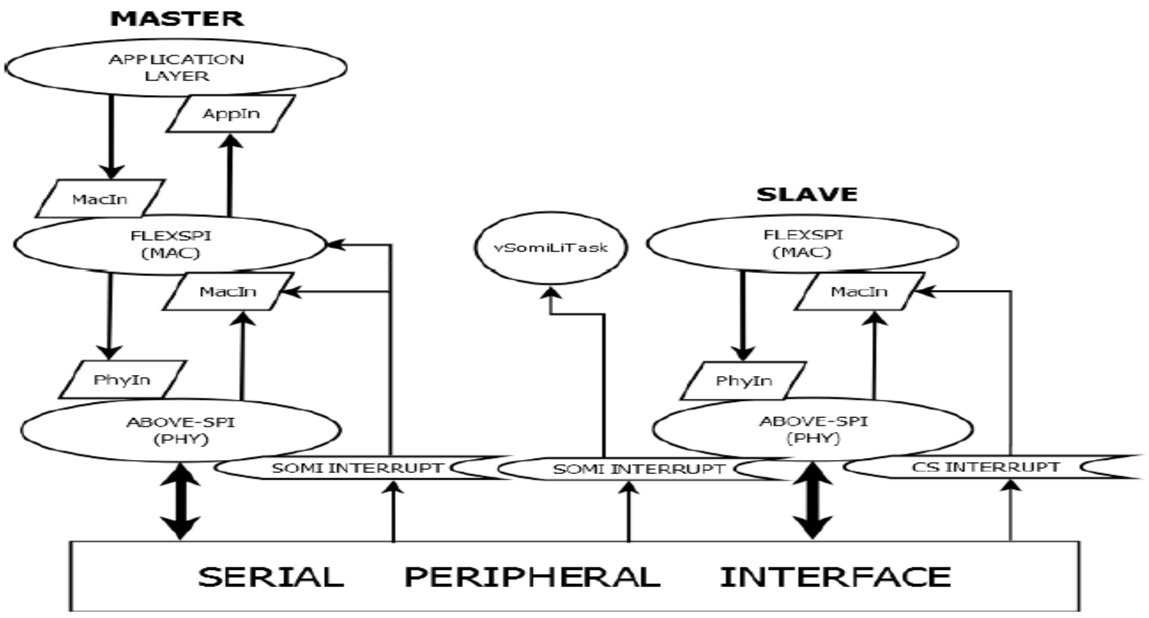


Figure 34. FlexSPI layered stack modified for SOMI-LI procedure.

```
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR(void){

 extern volatile structPlmeCSeventIndication_t *pVdataPlmeCSeventIndication;
  portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;
   switch(P1IV){
   case (0xA):
     P1IFG &= ~(1<<4); //event on CS
     xQueueSendFromISR(xQueueMacIn, (void *) &pVdataPlmeCSeventIndication , &
         xHigherPriorityTaskWoken);
     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
     break;
   case(0xE):
     P1IFG &= ~(1<<6); //event on SOMI
     xSemaphoreGiveFromISR(xSemSomiLi ,&xHigherPriorityTaskWoken);//somi-li activation
     portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
     break;
   default: break;
   }
}
```

Listing. Slave PORT1 Interrupt Service Routine (ISR).

As can be noted from listing, the detection of a falling edge on SOMI line causes the release of a semaphore, which in turn unblocks the parallel task *vSomiLiTask*. In this way, slave can start the operations required to confirm a malfunction in the link with the master, excluding a false alarm. *vSomiLiTask* parallel task, before confirming that link is broken, has to wait for a given amount of time, almost one second in this framework. Once this timeout expires, the slave checks again *SOMI* pin logical level: if it returns high, a false alarm has been detected;

if it remains low, the link is considered broken. LEDs have also been used to monitor the different phases of this procedure: the red LED turns on immediately after the task is entered. As reported in the implementation of this task in following listing, two LEDs are turned on or off according to the outcome of verification; at last, to perform reliable tests, when the link is considered broken, the slave scheduler is immediately suspended.

```
void vSomiLiTask( void *pvParameters ) {
  for(;;){
    xSemaphoreTake( xSemSomiLi, portMAX_DELAY );
    taskDISABLE_INTERRUPTS();
    turn_ledON(red);
    __delay_cycles(18000000); //wait time
    uint8_t trial = (P3IN & (1<<2))>>2;
    if(trial == 1){ //p3.2 analysis high
      turn_ledOFF(red);
    }
    else{
      turn_ledON(orange);
      vTaskSuspendAll ();//link is dead!
    }
    taskENABLE_INTERRUPTS();
  }
}
```

Listing: *vSomiLiTask* implementation.

From a hardware point of view, as reported in figure 35, the implementation of this procedure has been obtained by performing two different actions. The first one is the presence of a short circuit among SOMI pin and a GPIO pin of slave to enable interrupt capabilities. The second modification has been made so that *SOMI* line was not floating when bus had been voluntarily removed: a floating line, in fact, could cause spurious pulses eventually misinterpreted by the device. For this reason, a pull-down resistor has been placed between the SOMI line and GND: in this way, when the link gets broken, the line is marked with a low logical level.

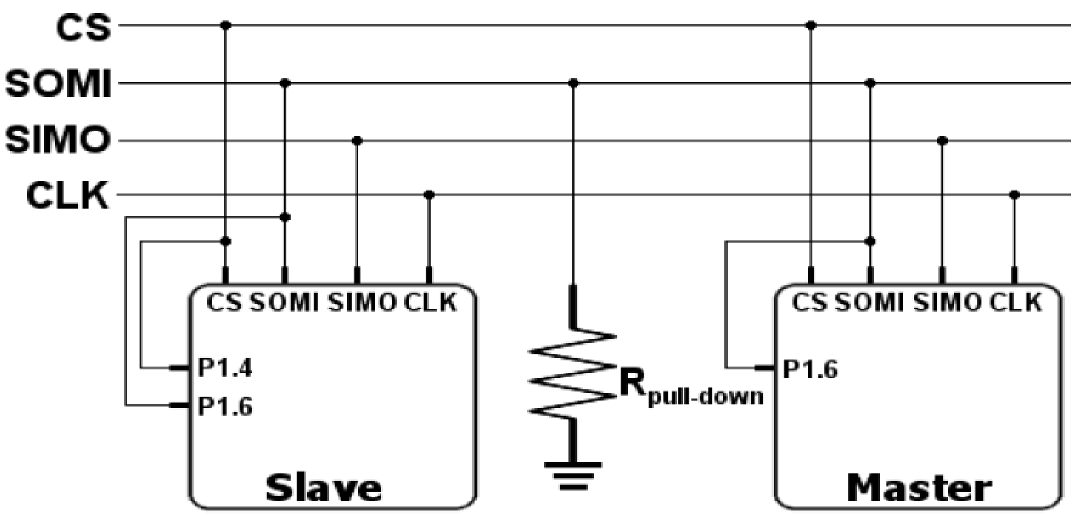


Figure 35. Electrical scheme of the experimental setup for the SOMI-LI procedure.

The employed resistor, equal to 130 k, ensures a correct logical level whether the bus link is present or not. However, it must be considered that its value affects the time the line needs to return high when a spurious pulse is detected: for this reason, the time the device must wait before declaring that the link is broken must be dimensioned according, among other things, to pulling resistors in play. The exposed scheme was used to realize the experimental setup reported in figure 36, where the pull-down resistor has been connected using a breadboard.

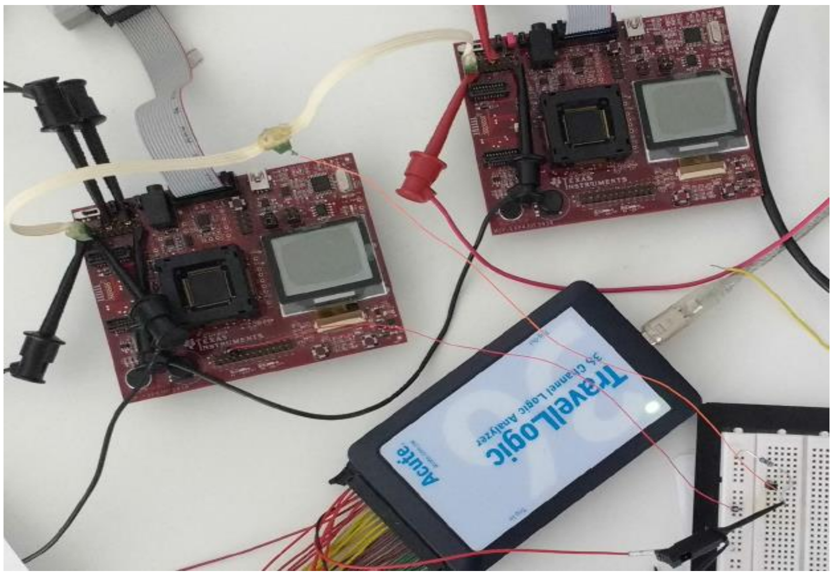


Figure 36. Realized experimental setup for the SOMI-LI procedure.

This procedure has been verified by manually causing the break of the link: the first time, the connector, used to plug all available SPI pins, has been removed and reinserted before timeout expired, verifying the false alarm recognition. The second time, instead, the connector was not reinserted after its removal: since the link remained broken, the orange LED turned on and the device stopped working. In this way, it has been possible to verify that the system coherently reacted both in case of a simulated false alarm and when the link is actually broken. It must be taken into account that real malfunctions or link interruptions may not always be so defined.

c. PSN Procedure: Code Design and Validation

PHY Speed Negotiation (PSN) is the procedure used by slaves to negotiate the bus speed with the master; in this way, it is possible to optimize data exchange by tailoring the bus speed to the different capacities of devices present on the bus. The master, in fact, can configure the clock speed according to the slave it wants to communicate with. The PSN procedure consists of a dense packet exchange between the master and a slave, as already seen in figure 25. That behavior can be obtained by expanding the layered stack as reported in figure 37: supposing the negotiation takes place with the first slave, MAC layers are expanded with an auxiliary task interacting with it. The slave that starts the procedure has also an application layer, used to simulate a high level request to change the speed used for data transfer.

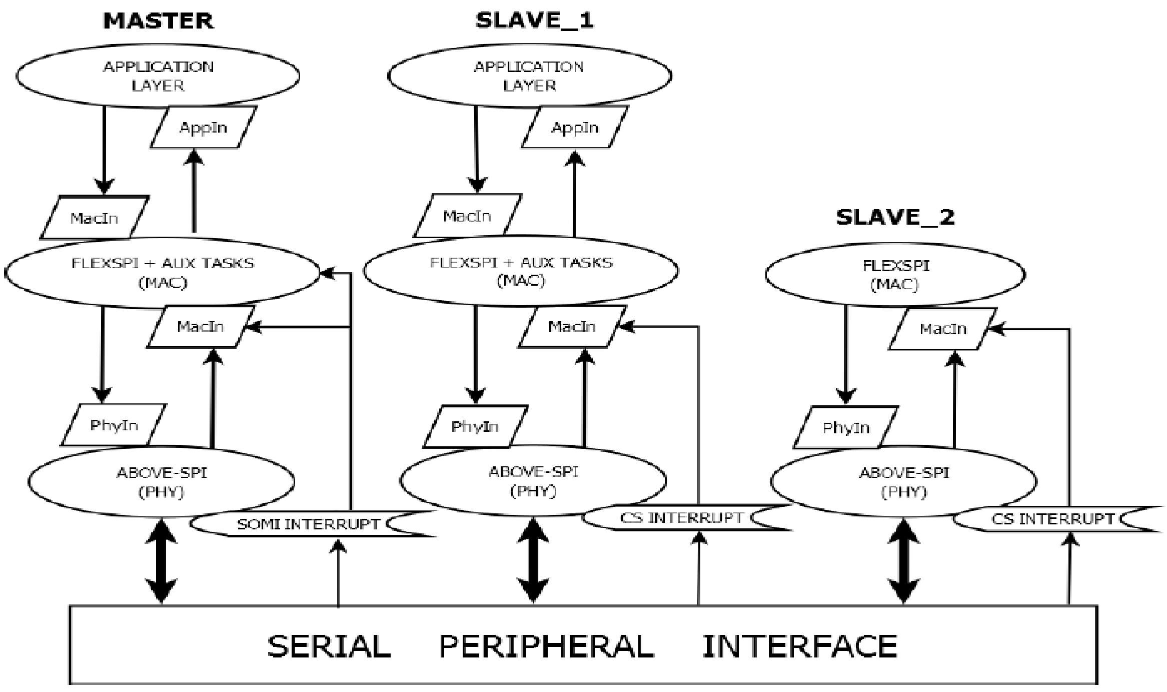


Figure 37. FlexSPI stacked layers expanded for PSN procedure.

Since this procedure is started from the slave, two things happen before the packet exchange of figure 25 takes place: the slave, thanks to its application layer, requests a new speed and triggers the generation of a SOMI pulse; the master, supposing that it knows who produced this pulse, sends a *POLLSLAVE* packet to listen to the slave request. The packet exchange used to perform PSN procedure exploits the finite state machines that set register *macState*, conveniently expanded for both the master and slave. The auxiliary task, called *vPsnSTask*, is responsible for updating the state of *macState*, assisting the MAC layer when necessary; this task has a different implementation whether the device is the master or a slave.

Master finite state machine is expanded as reported in figure 38a: as can be seen, its extension is backward compatible with the ping procedure. The presence of the auxiliary task in the MAC layer ensures that *SPEEDACK* packet is sent twice: in fact, after first acknowledgement, the master sets the new speed and tests the bus to verify that data exchanges with the required clock frequency are supported.

The *SPEEDREQ* packet sent by the slave to master has a payload containing the speed that the slave wants to be used for future data exchanges. However, in a generic scenario, the master may not always be able to grant slave request, and the decision has to be made according a specific criterion. Since the metric that the master uses deeply depends on the bus itself and on the application, in this framework, the master decides according a threshold: if the request is smaller than 3 MHz it can be granted, otherwise it is rejected. *Slave finite state machine* is extended as reported in figure 38b: the diagram is deeply expanded to provide support to all the different phases of the required packets exchange. As explained before, the transmission of the *SPEEDREQ* packet is triggered by the application layer, which activates the necessary operations to request to the master the possession of channel; the master sends *POLLSLAVE* packet and downloads the slave request.

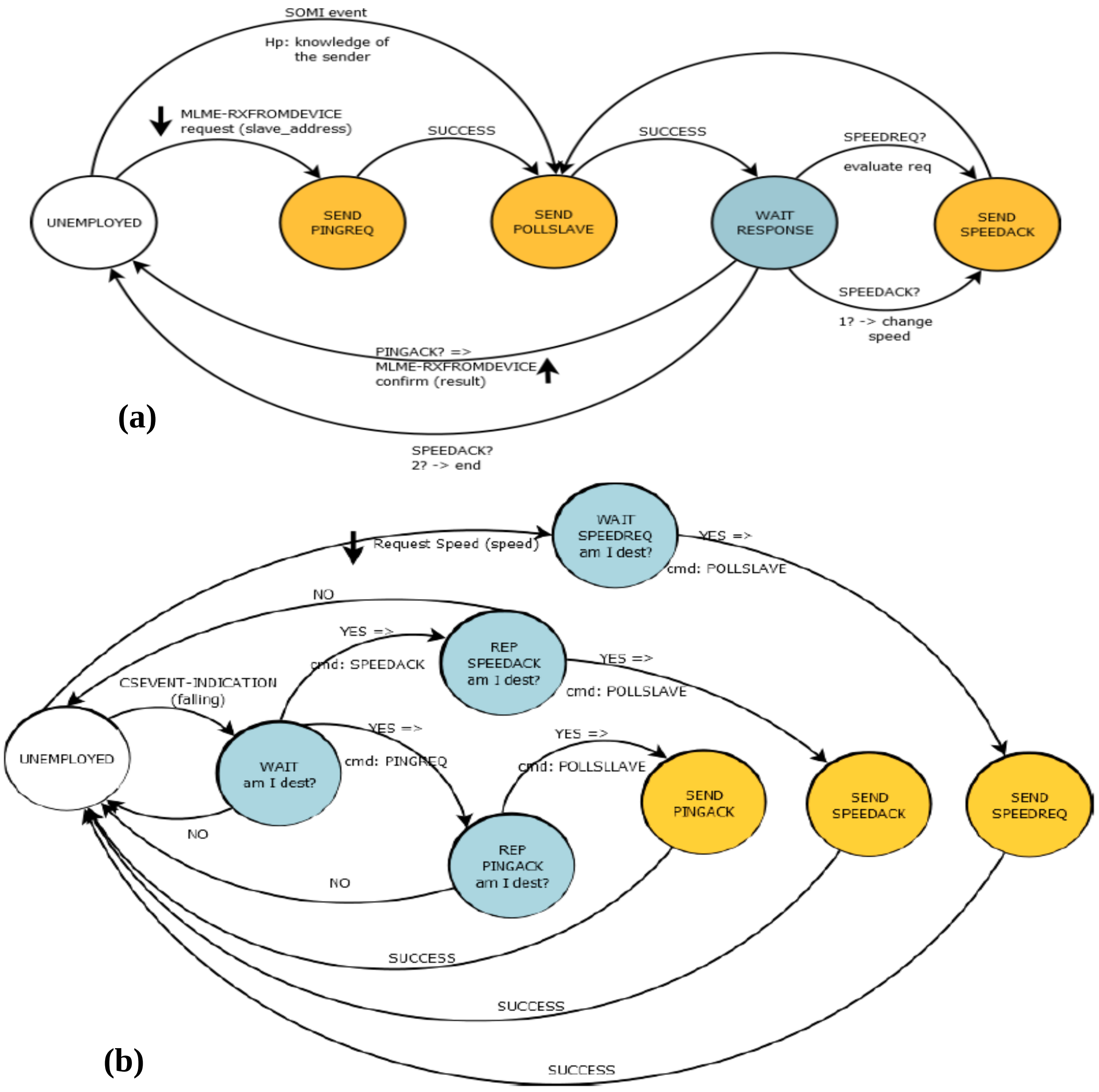


Figure 38. Expanded finite state machine of the master (a) and of the slaves (b).

The validation of this procedure has been obtained with only one device on bus, deactivating the application layer of the master. The master initializes a bus speed of ≈ 250 kHz, while the slave requests a clock speed equal to 1.5 MHz that will be accepted. The variation of bus speed can be first qualitatively evaluated by looking figure 39, which shows the overall packet exchange: the last three packets, sent with new speed, have a short duration, as can be deduced by the thinner strips referred to the clock pulses.

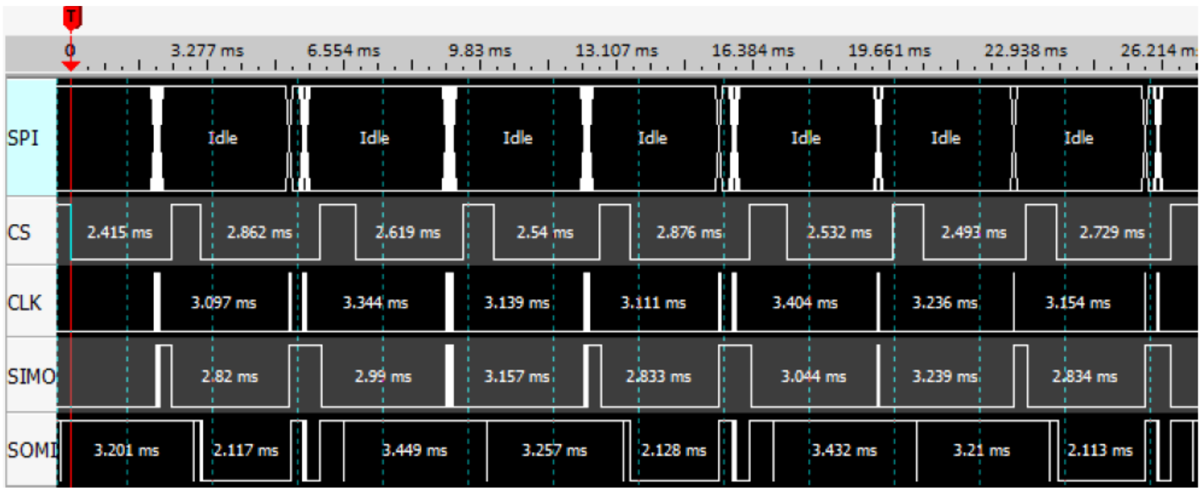


Figure 39. Signals exchanged during the PSN procedure.

The detail of all the exchanged packets transmitted to perform the procedure are reported in the following figures 40 and 41. After the slave *SOMI* pulse and *POLLSLAVE* packet, slave is authorized to transmit its *SPEEDREQ* packet; the payload, composed by 2 bytes, contains the desired speed. The master, using its parallel task, evaluates the request and, once found acceptable, sends a *SPEEDACK* packet having as a payload the speed that is going to be set.

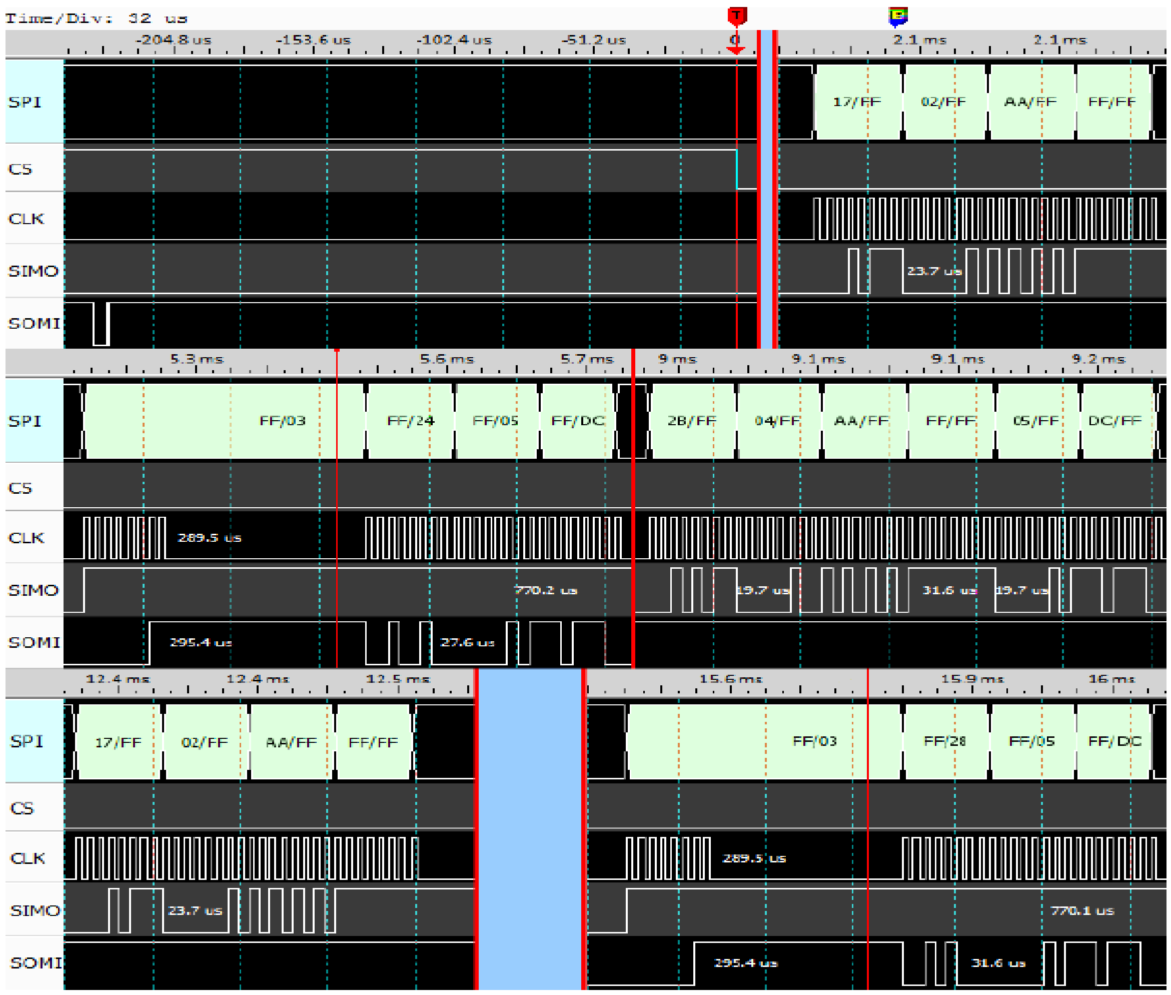


Figure 40. Time-domain details of the PSN packets exchanged with the old speed.

The master then sends a *POLLSLAVE* packet in order to retrieve *SPEEDACK* command from the slave. Once this exchange has correctly taken place, the master modifies bus speed and tests the channel. Figure 41 represents the data exchange that occurs with negotiated speed: because of the higher speed, resolution has been increased in order to be able to recognize the waveforms. The master starts the channel test by sending, once again, a *SPEEDACK* packet; after grating the channel to the addressed slave with *POLLSLAVE* command, the *SPEEDACK* reply of slave is downloaded. This exchange completes the procedure and the good outcome of all the exchanged packets is used by the master to verify that the speed is supported. At last, the change of speed is shown in figure 42, representing the detail of the two *SPEEDACK* commands that the master sends during the procedure. As described before, in fact, the master sends twice this packet to the slave, one to acknowledge the request and one to test the bus.

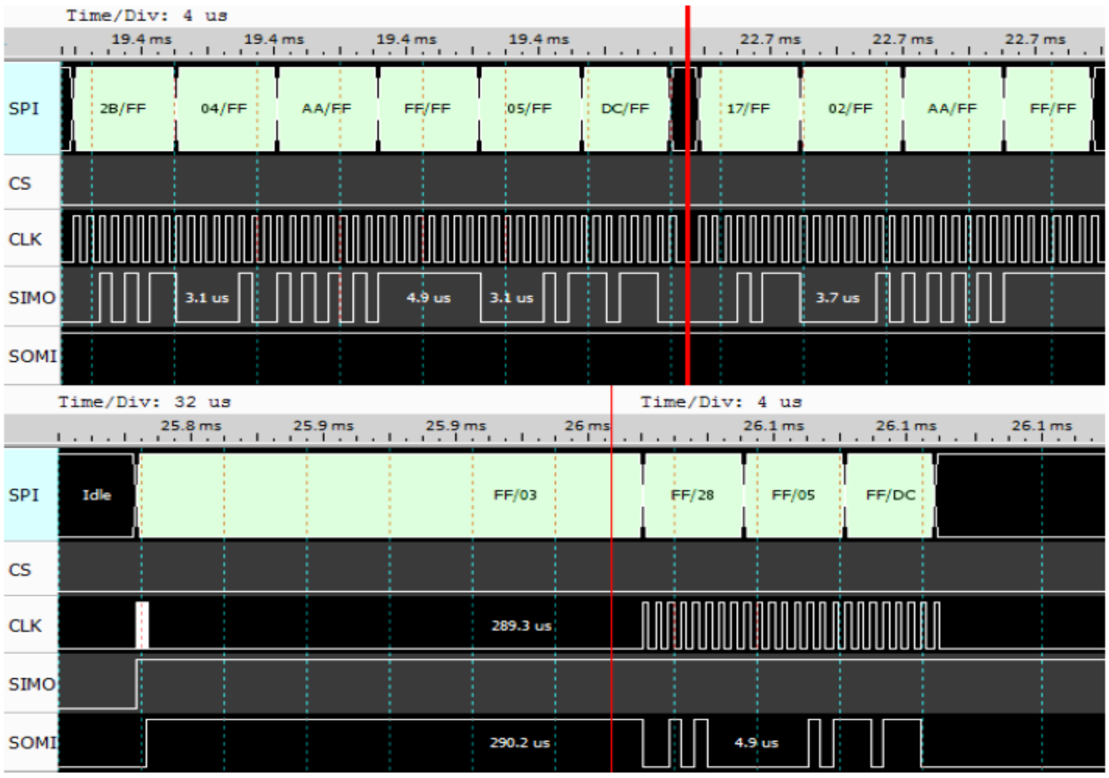
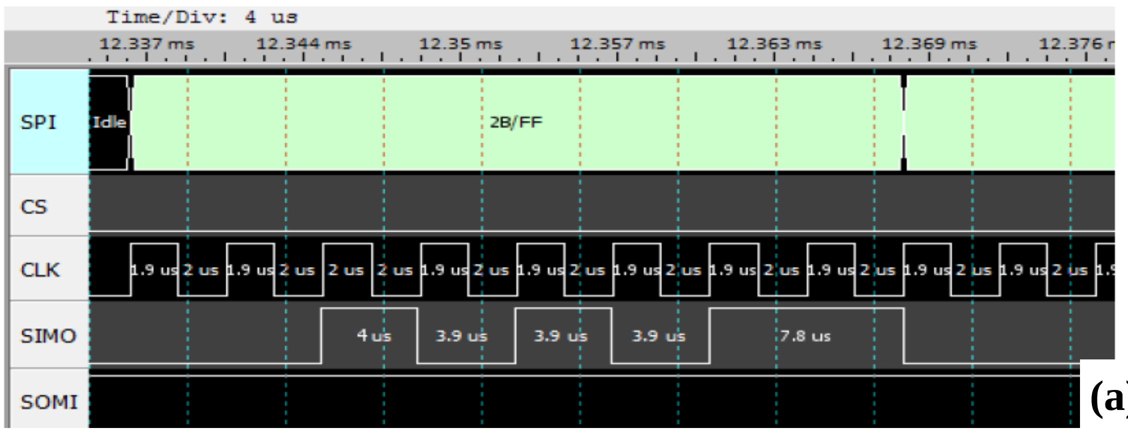


Figure 41. Time-domain details of the PSN packets exchanged with the new speed.



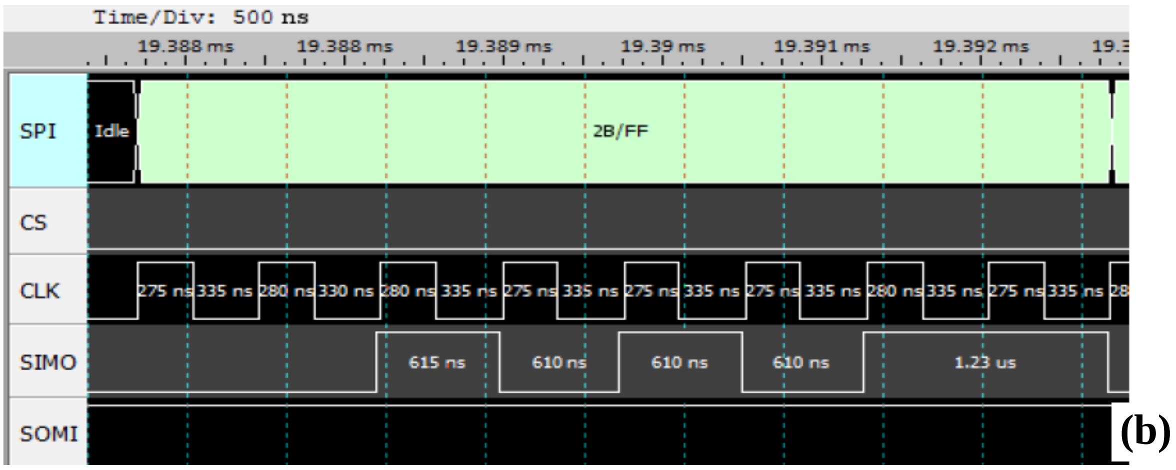
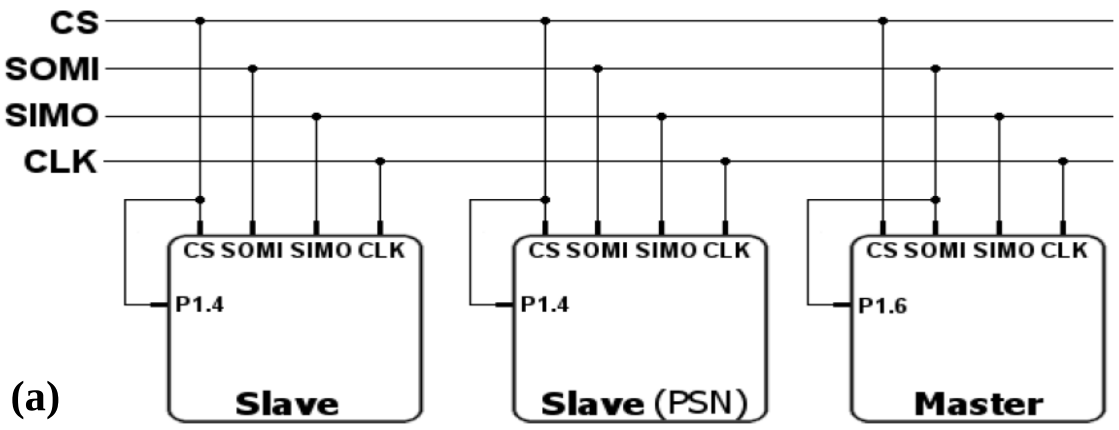


Figure 42. Comparison of the two bus speeds during PSN procedure, low (a) and high speed (b).

d. Example of Master smartness Improvement

A final test has been executed, aiming to test the benefits of FlexSPI in enhancing smartness of the devices that use this communication protocol to exchange data. This framework shows how the master can adapt data exchange speed according to addressed slave, becoming aware of the bus structure and reacting dynamically to changes. Both the ping and speed negotiation procedures have been used, triggering their activation with the buttons available on the experimenter board; MADSIS procedure keeps being used by the devices, too. The master, during its initialization phase, creates a table of the devices on the bus, associating to every address a fixed speed set to 250 kHz; a first ping procedure is then performed. After that, the slave used before performs the speed negotiation procedure, requiring a 1.5 MHz bus speed, and master updates coherently its table. The master then performs, once again, the ping procedure; this time, however, different speeds are used to communicate with the two devices. The scheme of experimental setup used for this test is represented in figure 43a. The master also possesses the necessary SOMI short circuit to perform MADSIS procedure; the slave requiring the new speed is the one on the center. This model has been followed to perform the connection between the devices, obtaining the setup shown in figure 43b.



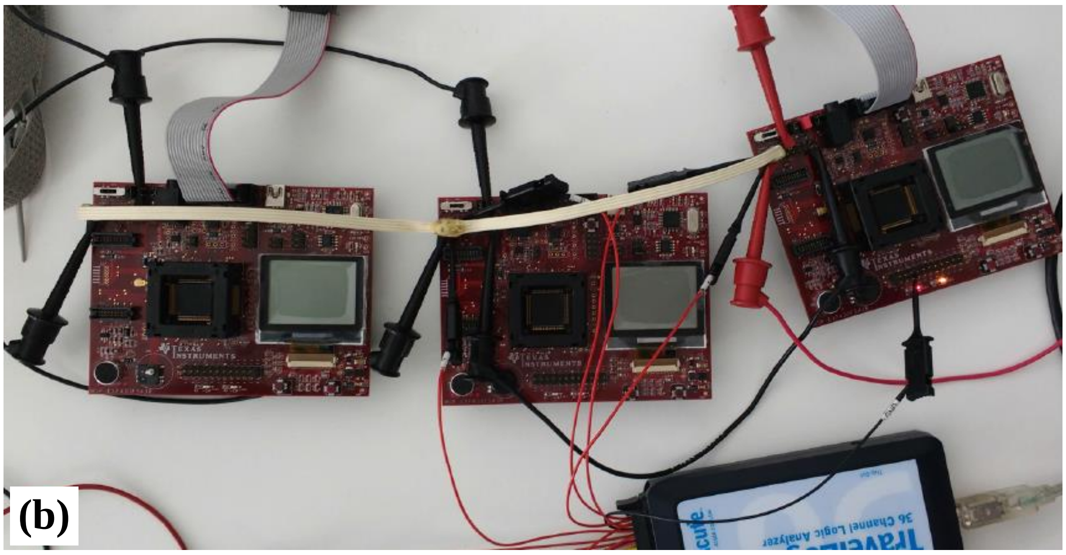


Figure 43. Scheme (a) and realized experimental setup for smartness improvement test (b).

As usual, exchanged signals have been collected using the logical analyzer. Figure 44 shows time intervals when the two ping procedures are performed, before and after slave PSN. By focusing on CLK line and looking on the width of the white stripe, it is possible to deduce that the master correctly reads and updates its table: a faster speed bus is, in fact, used for the first slave during second ping, changing coherently the clock speed when the second slave is polled.

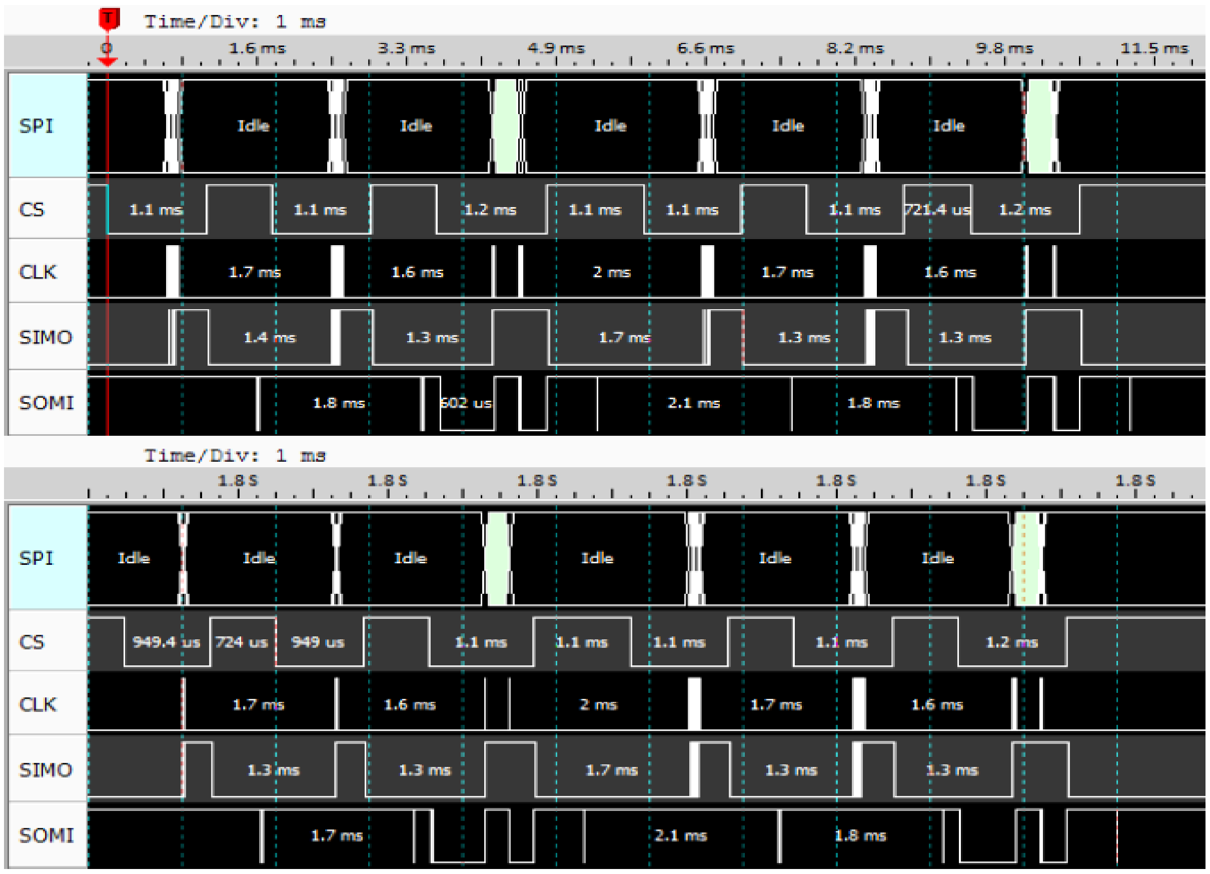


Figure 44. Detail of the two ping procedure performed before and after that the first slave negotiated a new speed.

All the presented procedures highlight the advantages of using a communication protocol based on a structured software architecture. It has been possible, in fact, to analyze advantages of a synchronization mechanism among the devices on bus, using extra signaling; it has also been observed how devices can be made "*channel aware*", constantly monitoring the bus and, at the same time, protecting themselves from false alarms. Thanks to speed negotiation, it is possible to build a bus made by devices deeply different one another: the master, in fact, can be made smart and adapt its behavior according to the task that must fulfill. These aspects are very adherent to the IoT principles proving how beneficial can be adopting this protocol when developing advanced smart objects. Obviously, nothing comes without a price: apart from software complexity, an investigation in terms of energy consumption has to be made in order to understand if the advantages deriving from this protocol can be obtained without increasing power consumption to levels harming the device energetic autonomy. This is the aim of future research work in which FlexSPI framework will be used to quantify the energy consumption.

VI. CONCLUSIONS

All the explained procedures, joint with its physical architecture, stress the potentiality of developed FlexSPI protocol, capable of providing a lot functions tailored for dynamic systems and optimizing communication sessions. The dynamic switch among the peripherals present in microcontrollers gives also the possibility to slaves to safely trigger events captured by the master, surpassing the rigid scheme of classic master-slave architectures in which only master is responsible for every action on the bus. The fixed number of wires, together with the push-pull data exchange provided by SPI, make possible to create simpler circuital layouts for devices with several components without renouncing to a high-speed throughput [17]. I^2C protocol, natural candidate for a comparison, although provides some variants of these procedures is limited by its RC connection, as explained in [1]. FlexSPI can be built like a MAC layer above the SPI bus to process all necessary pieces of information to perform the packet level addressing, using a stack having a layered architecture. This is idea followed in the firmware development to implement this communication protocol experimentally verifying that it is possible to obtain a shared push-pull based bus. Some of the discussed procedures have been also implemented to directly check their working mechanisms and advantages. In order to give some context to obtained results, the developed firmware has been also used to make a direct comparison with I^2C, providing an indication on which devices could have more benefits from this protocol and how its implementation can be improved.

REFERENCES

[1]. P. Visconti, G. Giannotta, R. Brama, P. Primiceri, A. Malvasi, "Features, operation principle and limits of SPI and I^{2}C communication protocols for smart objects: a novel SPI-based hybrid protocol especially suitable for IoT applications"; Int. Journal on Smart Sensing and Intelligent Systems, ISSN 1178-5608, Vol. 10 (Issue 2), pp. 262 - 295 (2017).

[2]. K. Ashton, "That 'Internet of Things' Thing", RFID Journal, Jun 2009.

[3]. G. Kortuem, F. Kawsar, D. Fitton, V. Sundramoorthy, "Smart objects as building blocks for the internet of things," Internet Computing, IEEE, vol. 14 (Issue 1), pp. 44–51 (2010).

[4]. M. Jyothi, L. Ravi Chandra, M. Sahithi, S. Daya Sagar Chowdary , K. Rajasekhar, K. Purnima; "Implementation of SPI Communication Protocol for Multipurpose Applications with I^{2}C Power and Area Reduction", International Journal of Engineering Research and Applications - IJERA, ISSN: 2248-9622, Vol. 2, Issue 2, pp.875-883 (2012).

[5]. G. Khurana, U. Goyal "An Insight Comparison of Serial Communication Protocols", International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE), Vol. 2, Issue 3, ISSN: 2277 – 9043, pp. 308 - 313 (2013).

[6]. R. Barry, "Using the FreeRTOS Real Time Kernel - ARM Cortex", M3 Edition. *Real Time Engineers LTD* (2010).

[7]. P.Visconti, P. Primiceri, C. Orlando; "Solar Powered Wireless Monitoring System of Environmental Conditions for Early Flood Prediction or Optimized Irrigation in Agriculture", ARPN Journal of Engineering and Applied Sciences, Vol. 11 (Issue 7), pp. 4623-4632, (2016).

[8]. P.Visconti, C.Orlando, P. Primiceri; "Solar Powered WSN for monitoring environment and soil parameters by specific app for mobile devices usable for early flood prediction or water savings". IEEE 16th Int. Conf. on Environment and Electrical Engineering EEEIC, DOI: 10.1109/EEEIC.2016.7555638, SCOPUS = 2-s2.0-84988418455 (2016).

[9]. P.Primiceri, P.Visconti, A.Melpignano, A.Vilei, G.M.Colleoni; "Hardware and software solution developed in ARM mbed environment for driving and controlling DC brushless motors based on ST X-NUCLEO development boards". Int. Journal on Smart Sensing and Intelligent Systems, Vol. 9 (3), pp. 1534 - 1562, SCOPUS eid = 2-s2.0-84992448380 (2016).

[10]. P. Visconti, R. Ria, G. Cavalera; "Development of smart PIC–based electronic equipment for managing and monitoring energy production of photovoltaic plan with wireless transmission unit". ARPN Journal of Engineering and Applied Sciences, Vol. 10 (Issue n. 20), pp. 9434 - 9441, http://www.arpnjournals.com/jeas/volume_20_2015.htm, (2015).

[11]. P. Visconti, A. Lay-Ekuakille, P. Primiceri, G. Cavalera; "Wireless Energy Monitoring System of Photovoltaic Plants with Smart Anti-Theft solution integrated with Household Electrical Consumption's Control Unit Remotely Controlled by Internet". Int. Journal on Smart Sensing and Intelligent Systems, Vol. 9 (Issue 2), pp. 681 – 708 (2016).

[12]. P.Visconti, P. Primiceri, P.Costantini, G.Colangelo, G. Cavalera, "Measurement and control system for thermo-solar plant and performance comparison between traditional and nanofluid solar thermal collectors"; Int. Journal on Smart Sensing and Intelligent Systems, Vol. 9 (Issue 3), pp. 1220 – 1242, http://s2is.org/Issues/v9/n3/papers/paper3.pdf (2016).

[13]. P.Visconti, R.de Fazio, P.Primiceri, A.Lay-Ekuakille; "A solar-powered White LED-based UV-VIS spectrophotometric system managed by PC for air pollution detection in faraway and unfriendly locations". Int. Journal on Smart Sensing and Intelligent Systems, Vol. 10 (1), pp. 18 - 48, http://s2is.org/Issues/v10/n1/, SCOPUS = 2-s2.0-85014108211 (2017).

[14]. P. Visconti, A. Lay-Ekuakille, P. Primiceri, G. Ciccarese, R. de Fazio; "Hardware design and software development for a White LED-based experimental spectrophotometer managed by a PIC-based control system" IEEE Sensors Journal, ISSN: 1530-437X, Vol. 17 (Issue 8), pp. 2507 - 2515, DOI: 10.1109/JSEN.2017.2669529 (2017).

[15]. P. Primiceri, P.Visconti; "Solar-powered LED-based lighting facilities: an overview on recent technologies and embedded IoT devices to obtain wireless control, energy savings and quick maintenance" Journal of Engineering and Applied Sciences ARPN, Vol. 12 (Issue 1), pp. 140-150 (2017).

[16]. T. Leal del Río, G. Juarez Gracia, L. N. Oliva Moreno; "Implementation of the communication protocols SPI and I^2C using a FPGA by the HDL-Verilog language", Research in Computing Science, Vol. 75, pp. 31–41 (2014).

[17]. R. Brama, P. Tundo, A. Della Ducata, A. Malvasi, "An Inter-Device Communication Protocol for Modular Smart-Objects", Proc. of 2014 IEEE World Forum on Internet of Things (WF-IoT) , pp. 422-427 (2014).

[18]. H. Ghayvat, A. Nag, N. K. Suryadevara, S.C. Mukhopadhyay, X. Gui and J. Liu; "Sharing research experiences of WSN based Smart Home", International Journal on Smart Sensing and Intelligent Systems, Vol. 7 (4), pp. 1997-2013 (2014).

[19]. Book: "Wearable Electronics Sensors for Safe and Healthy Living", Vol. 15 "Smart Sensors, Measurement and Instrumentation", editor: Subhas Chandra Mukhopadhyay, ISSN: 2194-8402, DOI 10.1007/978-3-319-18191-2, Springer Int. Publishing Switzerland (2015).