Hardware support for memory protection in sensor nodes

Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione, Università di Pisa, via G. Caruso 16, 56126 Pisa, Italy
E-mail: l.lopriore@iet.unipi.it

Abstract — With reference to the typical hardware configuration of a sensor node, we present the architecture of a memory protection unit (MPU) designed as a low-complexity addition to the microcontroller. The MPU is aimed at supporting memory protection and the privileged execution mode. It is connected to the system buses, and is seen by the processor as a memory-mapped input/output device. The contents of the internal MPU registers specify the composition of the protection contexts of the running program in terms of access rights for the memory pages. The MPU generates a hardware interrupt to the processor when it detects a protection violation. The proposed MPU architecture is evaluated from a number of salient viewpoints, which include the distribution, review and revocation of access permissions, and the support for important memory protection paradigms, including hierarchical contexts and protection rings.

Keywords: access right, privileged mode, memory protection, sensor node.

1. INTRODUCTION

In the architecture of a sensor node, stringent limitations exist in terms of hardware complexity [1], [17]. These include the absence of a memory management unit for virtual to physical address translation, and the lack of processor support for the two traditional execution modes, a kernel, privileged mode and a user mode with memory access limitations [6], [9], [13]. This means that the operating system kernel and the user programs share a single address space [2]. Each program has unlimited access to the whole primary memory, as well as to all hardware resources, including input/output devices, timers, sensors and actuators.

Owing to the lack of memory protection, program errors and illegitimate program behavior have a potential for corrupting system integrity [7]. The kernel is especially vulnerable. Every program is in a position to access the information items strictly reserved for the kernel, e.g. the cryptographic keys. Consequently, an erroneous or deliberately harmful program can disrupt the whole node [12], [21]. In a sensor network, programs are not prevented from using the cryptographic keys of the network protocols [23]; damages may ensue that cross the node boundaries and span a possibly large network fraction. The problem of programming errors is exacerbated by the fact that the writing of application software for sensor nodes is a quite complex task, owing to the requirements for real-time response and support for concurrency, the necessity to comply with different classes of sensors and actuators, and the stringent limitations in terms of memory space, processing power and energy consumption [4], [16].

Partial solutions to these problems have been proposed in the past. In [22] software execution in isolated compartments is obtained by extending the microcontroller with ad-hoc hardware in the form of a *memory protection unit* (*MPU*) that partitions the memory into up to 128 segments of variable size. The MPU is interposed between the processor and the memory modules; it generates an interrupt to the processor in case an access violation is detected. The variable segment size increases MPU hardware complexity. No privileged mode of execution is supported. In [5] a form of protection against control flow attacks is proposed that is based on utilization of a separate stack, called the *return stack*, contained in a memory location different from the normal execution stack. The return stack is reserved for storage of the return addresses. Protection of the return stack against accidental or deliberately harmful alterations relies on dedicated hardware, and implies a modification of the call and return machine instructions. In [7] the problem of preventing memory corruption from erroneous applications is dealt with in a hardware/software co-design approach to memory protection that relies on processor core enhancements, e.g. in the implementation of the store, call and return instructions. A memory map checker is interposed between the processor and the data memory, and is aimed at validating the memory accesses at the hardware level. Protection domains are supported, but no mechanism prevents an erroneous module from corrupting its own state, which resides completely within a single protection domain.

In contrast, with reference to the typical hardware configuration of a sensor node, this paper presents the architecture of a memory protection unit that has been designed by taking the following security and architectural requirements into account: (i) the complexity of the MPU architecture should be low, so that the cost of the additional hardware is kept to a minimum (e.g. much lower than that of a traditional memory management unit with the address translation portion removed); (ii) the inclusion of the MPU into the existing system should imply no modification of the architecture of the microcontroller, e.g. the instruction set; (iii) a privileged mode should be provided to encapsulate the memory area reserved for the kernel and make this area inaccessible to user programs; and (iv) a separation of the memory access privileges of different portions of the same software module should be supported.

In our protection model, the primary memory is partitioned into fixed-size pages. Memory protection is based on protection domains specifying collections of access rights for the memory pages. In the privileged mode, reserved for the kernel, memory protection is disabled. This means that the kernel routines are allowed to access the whole primary memory.

The rest of this paper is organized as follows. Section 2 presents the architecture of the MPU. A set of system commands is introduced, the *MPU commands*, which allow the running

program to modify the composition of the protection domains in a strictly controlled fashion. Special attention is paid to the transition to the privileged mode that ensues when an interrupt handling routine is executed. Section 3 discusses the proposed MPU architecture from a number of salient viewpoints, including the distribution, review and revocation of access permissions, and the support for important memory protection paradigms, including hierarchical contexts and protection rings. Finally, Section 4 gives concluding remarks.

2. MPU ARCHITECTURE

As anticipated in Section 1, in our protection model, fixed-size pages are the basic unit of memory protection. A protection domain is expressed in terms of a collection of access rights for the memory pages; the possible access rights are READ and WRITE. When a program is running, it is assigned a *protection context* defined in terms of a set of protection domains. The protection context limits the extent of the program activities in the primary memory to the access rights in the component domains. The MPU supports two protection contexts for each program, a *global* context and a *local* context. The global context states the memory pages that can be accessed by the program as a whole. The program is free to limit the extent of the global context to form a local context for each program component, e.g. a subroutine. The local context is defined in terms of a subset of the domains in the global context. Thus, the global context enforces protection at program level; a program cannot violate the boundaries of its own global context to access the private memory pages of the other programs or the system kernel. On the other hand, the local context makes it possible to restrict the activity of the given program component to a fraction of the pages in the global context. Local contexts will be defined by taking the *principle of least privilege* [19] into consideration, according to which each software component should be assigned the most restricted set of access rights that allows this software component to carry out its job successfully.

2.1. Protection domains

We shall refer to a typical microcontroller configuration featuring a simple processor and a few kilobyte primary memory (Figure 1). The primary memory space is logically partitioned into p fixed-size pages. The MPU is connected to the system buses. Each time the processor generates a memory address, the MPU captures this address and the specification of the memory access mode (for read or write). If the MPU detects that the running program does not hold an access privilege congruent with the intended access, it generates a hardware interrupt to the processor. Thus, an interrupt from the MPU corresponds to an *exception of violated protection*.

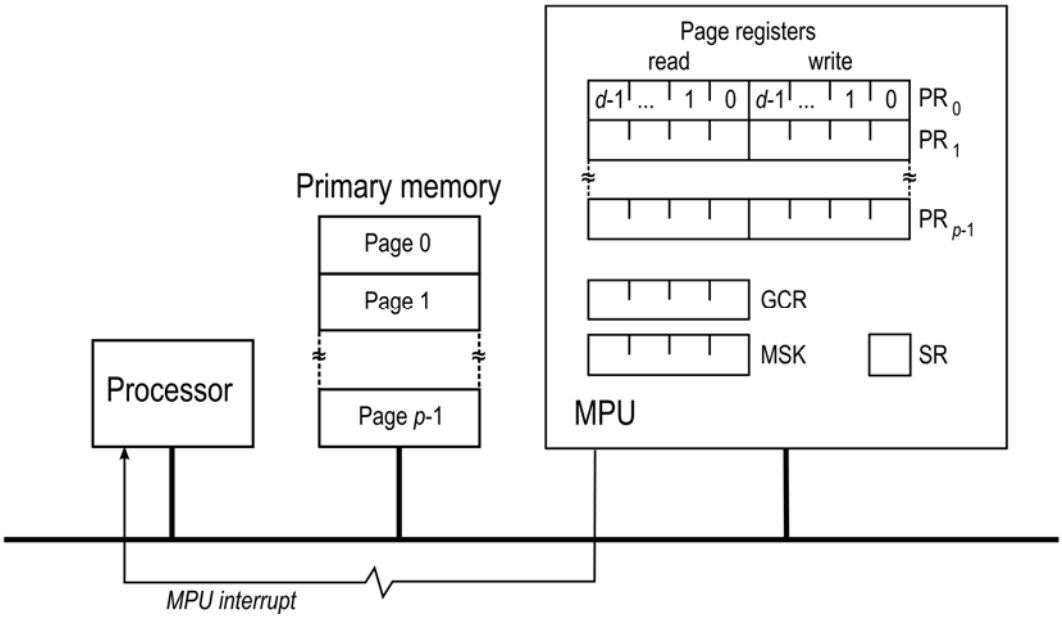

Figure 1. Hardware configuration featuring a processor, a primary memory, and a memory protection unit that is connected to the system buses and can generate hardware interrupts to the processor.

MPU interrupts are non-maskable, and have the highest priority.

The MPU hardware supports d protection domains by associating a *page register* with each memory page. Figure 2 shows the configuration of the page registers in a system featuring a three-page primary memory and four domains. Page register PR_i associated with page i is partitioned into two protection fields, corresponding to the two access rights, READ and WRITE. In the read protection field, the i-th bit, if asserted, specifies that access right READ is included in the i-th protection domain. This is similar to the write protection field for access right WRITE. It follows that the composition of the i-th domain is expressed by the contents of the i-th bit of the two page protection fields of all page registers.

In the configuration of Figure 2, domain D_0 includes access right READ for page 0. Domain D_1 contains both access rights READ and WRITE for page 1. Domain D_2 contains both access rights READ and WRITE for page 2. Domain D_3 contains access right READ for the three pages.

A protection context is a set of one or more protection domains. At any given time, an MPU register, the *global context register* (GCR), specifies the composition of the global context of the program running at that time. The size of GCR is d bits; if the i-th bit is set, then domain D_i is part of the global context.

A further MPU register, the *mask register* (MSK), specifies the composition of the local context in terms of a subset of the domains in the global context. The program component being executed by the processor can actually use these domains for successful memory access. Domain D_i that is part of the global context is also part of the local context only if the i-th bit of MSK is set. Thus, the composition of the local context is specified by the result of the bitwise

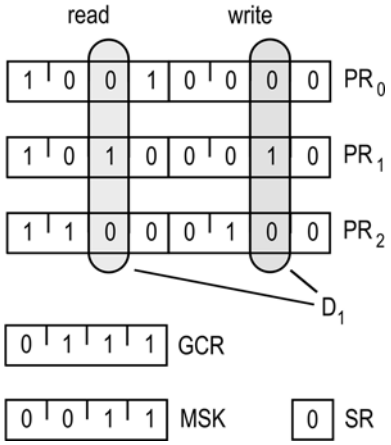

Figure 2. An example of configuration of the MPU registers.

AND of the contents of registers GCR and MSK. The mask register can be used to limit the access permissions held by a component of a given program, e.g. a subroutine, with respect to the access permissions held by the program as a whole in its global context. As will be illustrated in detail in subsequent Section 3.2, the configuration of GCR is defined for each program in the system configuration phase, whereas MSK is configured under explicit program control.

In the configuration of Figure 2, bits 0 to 2 of GCR are asserted and bit 3 is cleared; this means that the global context includes domains D_0 to D_2. In MSK, bits 0 and 1 are asserted and the other bits are cleared. It follows that domain D_2 is excluded from the local context. Thus, the current program component can only take advantage of the access rights in domain D_0 and D_1, i.e. access right READ for page 0, and access rights READ and WRITE for page 1.

Let us now suppose that the running program generates a memory access. The corresponding memory address from the processor is partitioned into the index i of the referenced page and an offset in this page. Inside the MPU, page register PR_i associated with page i is accessed, and the protection field corresponding to the type of the access (read or write) is considered. The bitwise AND of the contents of this protection field, of global context register GCR, and of mask register MSK is evaluated. If the result is 0, the access right required to accomplish the access is lacking; the MPU raises an exception of violated protection in the form of an interrupt to the processor.

2.2. The privileged mode

The MPU has two operating modes, privileged (kernel) and user. In the user mode, the running program is allowed to reference only those memory pages for which it holds access rights, according to the composition of the protection domains that form the local context, as is specified by the MPU registers. In the privileged mode, all the memory pages can be unconditionally accessed for both read and write; the exceptions of violated protection are inhibited.

A 1-bit register of the MPU, the *status register* (SR), specifies whether the MPU is in the privileged mode (SR = 1) or in the user mode (SR = 0). SR is part of the MPU interface, which is memory-mapped. No domain includes access permissions for the corresponding page, which is reserved for the kernel. It follows that if a user program tries to set SR, the MPU raises an exception of violated protection. In fact, SR is set as part of the actions involved in the processing of an interrupt.

The input/output devices are memory-mapped. The processor accesses the internal registers of these devices at fixed memory addresses. A user program can read or modify the contents of these registers only if it holds access permissions for the corresponding memory pages. Of course, these registers can be freely accessed by the kernel, when the MPU is in the privileged mode.

2.3. Interrupts

When the processor accepts an interrupt, it saves certain information onto the stack, including the program counter as a minimum; it disables all interrupts, and then accesses the interrupt vector to read the address of the next instruction. This is the address of the routine handling the interrupt.[1] Execution of this routine proceeds as follows:

1. The value of status register SR is read from the MPU and is pushed onto the stack. This action causes MPU to generate an interrupt that sets a flag inside the processor, which we shall call the *MPU flag*,[2] and remains pending, as all interrupts are disabled.
2. Register SR is set, thereby causing the MPU to enter the privileged mode.
3. The MPU flag is cleared, thereby cancelling the pending MPU interrupt.
4. The interrupt request is processed.
5. The previous value of SR is restored with quantities extracted from the stack.
6. A return-from-interrupt instruction is executed, which enables all interrupts and restores the value of the program counter, so that the interrupted software continues with the same status as before the interrupt.

Thus, during the interrupt request processing, the MPU will be in the privileged mode. In this phase, the whole primary memory can be accessed, including the internal MPU registers and, for instance, the memory-mapped interfaces of the input/output devices, as is required for

[1] Alternatively, the interrupt vector contains an instruction, usually a jump, and the processor executes this instruction as a consequence of the interrupt. In this case, the jump will lead to the interrupt handling routine.
[2] The MPU flag, if asserted, indicates that an MPU interrupt request is pending; it is cleared by the hardware when the MPU interrupt request is honored, and can be cleared by the software.

the processing of the interrupts generated by these devices.[3]

We wish to remark that our MPU design implies no modification of the microcontroller architecture. If this is not a requirement, and modifications are acceptable, most of the actions enumerated above can be transferred to the hardware. This is the case, for instance, for the saving of the value of SR onto the stack when an interrupt is accepted, and the restoring of the pervious value of SR with quantities taken from the stack, which can be incorporated into the return-from-interrupt instruction.

System calls

The running program generates a system call by loading the system call number where the kernel expects it, e.g. in a general register, and then issuing a software interrupt.[4] Execution of the routine handling the interrupt proceeds as has been illustrated above. It is worth noting that a program cannot execute an interrupt handling routine using a subroutine call rather than a system call, thereby bypassing the interrupt mechanism. In fact, these routines are stored in memory pages that are reserved for the kernel. No domain includes access rights for these pages, which can only be accessed when the MPU is in the privileged mode; the interrupt mechanism guarantees a transition of the MPU to the privileged mode.

2.4. MPU commands

The protection system defines a small set of commands, the *MPU commands*, which allow programs to modify the contents of the MPU registers in a strictly controlled fashion. We shall now illustrate these commands and the actions involved in the execution of each of them.

Protection domains are ordered from the highest-level domain D_0 to the lowest-level domain D_{d-1}. For the given page, the highest-level domain that contains a given access right is called the *prevailing domain* for that access right. In the situation of Figure 2, with reference to page 0, page register PR_0 specifies that domain D_0 is the prevailing domain for access right READ, for instance. A program whose local context (specified by the contents of registers GCR and MSK) includes an access right for a given page in the prevailing domain, is free to grant this access right to, and revoke this access right from, any lower level domain.

In detail, command *grantAR*(*AR*, *i*, *s*) inserts access right *AR* for page *i* into domain D_s, provided that the prevailing domain D_r for *AR* and page *i* is part of the local context, and $r < s$.

---

[3] A software routine running in the privileged mode is free to access all the internal MPU registers, including the status register SR. By clearing SR, the routine may well switch to the user mode; however, returning to the privileged mode is impossible, as SR is now inaccessible.

[4] If the instruction set of the processor does not include a system call instruction, a software interrupt can be generated by executing an instruction that causes an external device to produce a hardware interrupt.

Execution of this command sets bit *s* of the protection field that corresponds to *AR* in page register PR_i associated with page *i*.

Command *revokeAR*(*AR*, *i*, *s*) eliminates access right *AR* for page *i* from domain D_s, provided that the prevailing domain D_r for *AR* and page *i* is part of the local context, and $r < s$. Execution of this command clears bit *s* of the protection field that corresponds to *AR* in page register PR_i associated with page *i*.

Finally, command *setMask*(*msk*) replaces the contents of register MSK with quantity *msk*. As will be shown in Section 3.2, the running program can use this command to restrict the extent of the local context with respect to the global context for specific program components, e.g. a subroutine.

The MPU commands are designed to be fully implemented at the software level, by kernel routines. Programs will execute them by taking advantage of the system call mechanism, which has been illustrated in Section 2.3. Accesses to the memory-mapped interface of the MPU, required to modify the contents of the internal MPU registers, are made possible by the transition of the MPU to the privileged mode, which ensues when a system call is processed.

2.5. Preemptive scheduling, and program switching

In a classical sensor node architecture featuring no support for a privileged mode and no form of memory protection, it is unreliable to implement a form of preemptive scheduling based on interrupts from a real-time clock [3]. In fact, in a system of this type, a program is free to access the clock and augment the duration of its own time slice indefinitely. In a system protected by our MPU, an action of this type is destined to fail. In fact, no user program holds access permission for the interface of the real-time clock, which is mapped to a memory page reserved for the kernel, and can only be accessed when the MPU is in the privileged mode.

When a program switch takes place (the current program releases the processor and a new program is assigned the processor), the contents of the MPU registers GCR, MSK, and SR are saved into a kernel table; these registers are filled with quantities taken from this table for the new program, thereby restoring the global context, the local context and the privileged mode of the new program. On the other hand, the contents of the page registers are not program-specific. Instead, these contents are shared by all programs; they do not vary on the occurrence of a program switch.

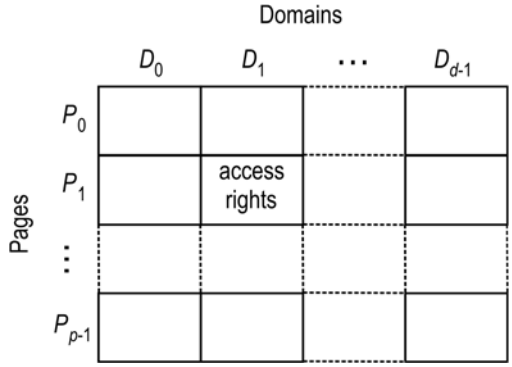

Figure 3. Configuration of the access matrix featuring a row for each page and a column for each protection domain. The generic matrix element specifies the access rights included in the domain in the corresponding column for the page in the corresponding row.

3. DISCUSSION

3.1. The protection model

In a well-known protection model, the protection state is depicted in the form of a matrix, called the *access matrix* [11], [15]. With reference to our protection system, Figure 3 shows the configuration of the access matrix for p pages and d domains. Row i is associated with page P_i, $i = 0, 1, \ldots, p - 1$, column j is associated with domain D_j, $j = 0, 1, \ldots, d - 1$, and element (i, j) specifies the access rights included in domain D_j for page P_i. A protection context corresponds to a set of one or more columns.

In the MPU, the page registers form our hardware-level implementation of the access matrix. The j-th column of the access matrix, corresponding to domain D_j, is represented by the j-th bit of the two protection fields, the read field and the write field, of all page protection registers (see Figure 1). The i-th row of the access matrix, corresponding to page i, is represented by page register PR_i, which specifies the access rights included in each domain for that page. Finally, registers GCR and MSK together specify the composition of the local context; if the j-th bit is set in both these registers, then the j-th domain is part of the local context.

In the rest of this section, we shall consider two important context organizations, hierarchical contexts and protection rings. We shall show that these organizations are well supported in the protection environment defined by the MPU.

Hierarchical contexts

In a hierarchical context organization, the contexts form a tree structure in which each context that is not a tree leaf inherits the access rights of its child contexts, recursively. In the MPU, we take advantage of register GCR to support hierarchical contexts, as follows. We reserve a single protection domain for each context. The protection domain reserved for a given

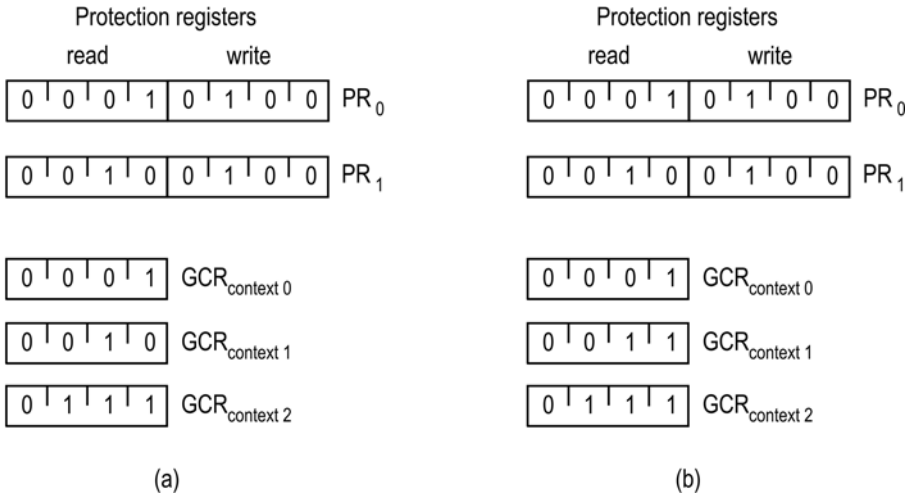

Figure 4. Configuration of the page registers and the global context register for: (a) hierarchical contexts; and (b) protection rings.

context contains the access rights associated with that context, which are not inherited from the child contexts. In the GCR configuration for a given context, we set the bit corresponding to the domain reserved for that context; furthermore, if the context has child contexts, we set the bits that are asserted in the GCR configuration for each of these child contexts.

With reference to a system configuration featuring two pages and four domains, let us refer to contexts 0 and 1 that share parent context 2, for instance. Suppose that context 0 includes access right READ for page 0, context 1 includes access right READ for page 1, and context 2 includes access right WRITE for both pages 0 and 1 and inherits access right READ for these pages from contexts 0 and 1. Figure 4a shows the corresponding configuration of the page registers and register GCR. Domain D_0 and D_1 are reserved for contexts 0 and 1, and domain D_2 is reserved for their parent context 2. D_0 includes access right READ for page 0, D_1 includes access right READ for page 1, and D_2 includes access right WRITE for both pages 0 and 1. In the configuration of GCR for context 0 we set bit 0, corresponding to domain D_0. Similarly, in the configuration of GCR for context 1 we set bit 1, corresponding to domain D_1. Finally, in the configuration of GCR for context 2 we set bit 2, corresponding to domain D_2, and bits 0 and 1 that are asserted in the GCR configuration for child contexts 0 and 1. So doing, context 2 inherits the access rights in the child contexts 0 and 1.

Protection rings

Protection rings are a form of hierarchical context organization in which each context that is not a leaf has a single child context, and a context at a given hierarchical level inherits the access rights of all the contexts at the lower levels [18], [20]. An organization of this type is usually represented in the form of concentric rings; the external ring corresponds to the least

privileged context, each inner ring corresponds to a context at a higher privilege level than the outer rings, and the central ring corresponds to the most privileged context.

In the MPU, we take advantage of register GCR to support a protection ring organization, as follows. We reserve a protection domain for each context. Protection domain D_i reserved for context i (corresponding to the i-th ring) contains the access rights associated with this context, which are not inherited from the contexts for the outer rings. In the GCR configuration for context i, we set bit i, corresponding to domain D_i, and bits $i - 1$, $i - 2$,…, 0, corresponding to domains D_{i-1}, D_{i-2},…, D_0, which are reserved for the contexts in the outer rings. In this way, context i inherits the access rights in these contexts.

In a system configuration featuring two pages and four domains, let us refer to a protection ring organization featuring three contexts, for instance. The least privileged context 0 includes access right READ for page 0; inner context 1 includes access right READ for page 1 and inherits access right READ for page 0 from context 0; and the most privileged context 2 includes access right WRITE for both pages 0 and 1 and inherits access right READ for these pages from contexts 0 and 1. Figure 4b shows the corresponding configuration of the page registers and register GCR. Domain D_0 is reserved for context 0 and includes access right READ for page 0, domain D_1 is reserved for context 1 and includes access right READ for page 1, and domain D_2 is reserved for context 2 and includes access right WRITE for both pages 0 and 1. In the configuration of register GCR for context 1, we set bits 0 and 1, for instance. In this way, context 1 inherits the access rights in outer context 0.

3.2. Global and local contexts

The initial composition of each domain, as well as the extent of the global context of each application program, will be defined in the system configuration phase. The initial state of the page registers, and of global context register GCR for each program, will be decided consequently. If, for instance, two or more programs are intended to share access to a given memory area, the global contexts of these programs will incorporate a domain featuring access rights for the pages that form this memory area. This means that the configuration of GCR for each of these programs will include this domain.

As seen in Section 2.4, command *setMask*() makes it possible to modify the contents of register MSK and, consequently, the composition of the local context. The programmer will add appropriate calls to *setMask*() to mask part of the domains out of the global context to form a local context tailored to the activity of each specific program component, e.g. a subroutine,

according to the principle of least privilege. So doing, the local context can limit the consequences of programming errors.

For instance, with reference to a given application program *G* and its subroutine *B*, let us consider a domain *D* that includes access rights for a few pages reserved for *B*. These pages are part of the address space of *G*, and consequently, the global context will contain domain *D*. The programmer will insert calls to command *setMask*() into the code of *G* to limit the extent of the local context to domain *D* when subroutine *B* is executed. If *B* generates an access attempt to a page that is not part of domain *D*, this is a symptom of a programming error; an exception of violated protection is raised, and the error is revealed.

We may conclude that our protection environment supports two different protection levels corresponding to different protection contexts. At the application program level, we have the global contexts. A given program cannot violate the boundaries of its own global context to access the memory areas reserved for the other programs or the kernel; an access attempt of this type is destined to be revealed by an exception of violated protection. Within the boundaries of a single application program, the distinction between the global context and the local context allows us to limit the consequence of programming errors and program faults. Here, we are concerned with safety rather than security [10], [14]. Of course, a deliberately harmful subroutine may well use *setMask*() to modify the contents of register MSK to amplify the extents of its own local context up to the limit of the entire global context.

3.3. Distribution, review and revocation of access permissions

As seen in Section 2.4, a program that holds an access right for a given page in the prevailing domain for that access right can take advantage of command *grantAR*() to distribute this access right to lower level domains. This mechanism supports page sharing between programs, for instance. Suppose that program G_1 holds access right *AR* for a given page *P* in domain D_r, and this domain is the prevailing domain for *AR*. G_1 will allow a different program G_2 to access *P* by granting *AR* to a lower level domain D_s that is part of the global context of G_2.

The revocation problem

A program that granted an access permission for a given page should be in a position to review its own intention and revoke this access permission from the recipient. In our system, the solution to this *revocation problem* is based on the ordering of domains into levels. A program that holds an access right for a given page in the prevailing domain for that access right can issue command *revokeAR*() to revoke this access right from a lower level domain.

In our previous example, program G_1 may take advantage of *revokeAR*() to revoke *AR* from

D_s. It is worth noting that G_2 cannot use *revokeAR*() to eliminate *AR* from the original owner G_1, as the level of D_s is lower than that of D_r.[5]

In spite of its simplicity, this access right revocation mechanism possesses a number of interesting properties. Revocation is [8]:

- *Selective*: a program can revoke a given access right from any subset of the domains to which it granted that access right. This result will be obtained by executing command *revokeAR*() on each domain in this subset.
- *Partial*: a program can revoke any subset of the access rights it granted to any given domain. As seen in Section 2.4, the access right subject to revocation is specified by argument *AR* of *revokeAR*().
- *Immediate*: if an access right is revoked from a given domain, this domain cannot take advantage of the revoked access right immediately past the time when revocation takes place.
- *Independent*: if the same access right was granted to a given program in two or more different domains, revocation can be limited to any subset of these domains.

3.4. Page size, and memory fragmentation

As seen in Section 2.1, the size of a protection register is $2 \cdot d$ bits, where d denotes the number of domains supported by the MPU. In a sensor node, the number of concurrent applications is usually quite limited; the layout of each application in memory is simple and almost static [22]. This means that if $d = 16$, for instance, more than a single domain can be reserved for each application, so that we can take advantage of mask register MSK to apply the principle of least privilege to specific software components, e.g. a subroutine (see Section 3.2). This can be a good trade-off between cost and usability, and is well suitable for typical low-power sensor node applications, even if a private memory (e.g. the stack) should be supported for each application. It should be clear that no domain needs to be reserved for the kernel, which is executed in the privileged mode, and consequently can access the whole memory; a page will be set aside for the kernel by simply excluding it from all domains.[6]

Of course, the partitioning of the primary memory into fixed-size pages is prone to forms

[5] Program G_2 cannot pass access right *AR* for page *P* to a third process, as G_2 holds *AR* in domain D_s which is not the prevailing domain for *AR*. This means that G_1 maintains full control over dissemination (and revocation) of *AR* throughout the system.

[6] Device drivers will be executed in the privileged mode, and the configuration of the page register array will exclude the pages reserved for the drivers from all domains. As a result, all drivers share a common memory space, and concurrent accesses to this space should be coordinated by taking advantage of the usual mechanisms for shared memory access. Communications between concurrent applications will take place via shared memory pages included in domains in common between these applications (see Section 3.2).

of memory fragmentation. A small page size reduces internal fragmentation but increases the number of page registers. As seen above, the size of a page register is quite limited, so small pages can be supported at low cost in terms of MPU hardware. For instance, if the memory size is 32 Kbytes, for a small page size of 256 bytes and 16 domains, we have a total of 128 pages with a memory requirement of 512 bytes for storage of the page registers (4 bytes for each page register); the total memory waste for internal fragmentation will be 2 Kbytes in the average (half page for each domain). The compiler/linker complex will be responsible for arranging and dimensioning domains to reduce external fragmentation. These are easy compiler tasks, which can be made largely transparent to the programmer. Placing new burdens on the compiler is a modern trend, which is now exploited in the solution of several architectural problems, e.g. instruction scheduling at compile time, translation lookaside buffer management, cache control and data prefetching.

As seen in Section 2.5, the contents of the page registers are not program-specific; these contents do not vary on the occurrence of a program switch. The page register array is not associative; instead, it is directly accessed by using the most significant bits of the address even in the presence of a large number of pages. No form of translation lookaside buffer is necessary even in the presence of a high number of page registers. These features are important to keep hardware complexity to a minimum. In fact, the MPU is definitively *not* a memory management unit with the address translation portion removed.

4. CONCLUDING REMARKS

With reference to the typical hardware configuration of a sensor node, we have considered the stringent limitations existing in terms of system functionalities supported by the hardware, and in particular, the lack of any form of memory protection and of a privileged execution mode. We have presented the architecture of a memory protection unit designed as a low-complexity addition to the microcontroller, and aimed at supporting these functionalities. The MPU is connected to the system buses, and is seen by the processor as a typical memory-mapped input/output device. The internal MPU registers specify the composition of the protection contexts of the running program in terms of access rights for the memory pages. The MPU inspects the system buses and generates a hardware interrupt to the processor when it detects a protection violation. The following is a brief summary of the main results we have obtained:

- No modification is necessary to the architecture of the microcontroller. In particular, the

instruction set of the processor is unaltered. We have obtained this important result by taking advantage of the interrupt mechanism to support the switch to the privileged mode, which is used to implement the MPU commands at software level.

- The duality of the global context and the local context corresponds to different protection requirements. The global context of a given program is a limitation of the memory areas that can be accessed by this program. It is aimed at protecting the other programs and the kernel from illegitimate access attempts. The local context is used within the program boundaries to restrict the memory areas that are accessible by specific program components to a subset of the global context, to limit the consequences of programming errors and program faults. Protection contexts allow an easy implementation of important memory protection paradigms. We have analyzed hierarchical contexts and protection rings in special depth.
- The MPU commands allow effective forms of distribution, review and revocation of access rights. A program that holds an access right in the prevalent domain can transfer this access right to a lower level domain, and subsequently revoke the access right. In this way, we support forms of program interaction and cooperation.
- The privileged mode allows us to protect input/output devices, sensors and actuators from illegitimate user program accesses. Forms of preemptive program scheduling based on interrupts from a real-time clock can be reliably implemented by encapsulating the memory-mapped interface of the clock into a memory page reserved for the kernel and inaccessible to user programs.

ACKNOWLEDGMENTS

The author thanks the anonymous reviewers for their insightful comments and constructive suggestions.

This work has been partially supported by the TENACE PRIN Project (Grant no. 20103P34XC_008) funded by the Italian Ministry of Education, University and Research.

REFERENCES

[1] M. Baunach, "Towards collaborative resource sharing under real-time conditions in multitasking and multicore environments," *Proceedings of the 17th IEEE Conference on Emerging Technologies & Factory Automation*, Cracow, Poland, September 2012.

[2] H. Cha *et al.*, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 2007, pp. 148–157.

[3] R. Chu *et al.*, "SenSmart: adaptive stack management for multitasking sensor networks," *IEEE Transactions on Computers*, vol. 62, no. 1 (January 2013), pp. 137–150.

[4] W. Dong *et al.*, "Providing OS support for wireless sensor networks: challenges and approaches," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4 (2010), pp. 519–530.

[5] A. Francillon, D. Perito, C. Castelluccia, "Defending embedded systems against control flow attacks," *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, Chicago, Illinois, USA, November 2009, pp. 19–26.

[6] R. Kumar, E. Kohler, M. Srivastava, "Harbor: software-based memory protection for sensor nodes," *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 2007, pp. 340–349.

[7] R. Kumar, A. Singhania, A. Castner, E. Kohler, M. Srivastava, "A system for coarse grained memory protection in tiny embedded processors," *Proceedings of the 44th Annual Conference on Design Automation*, San Diego, California, USA, June 2007, pp. 218–223.

[8] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6 (November 1979), pp. 575–586.

[9] V. C. Gungor, G. P. Hancke, "Industrial wireless sensor networks: challenges, design principles, and technical approaches," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 10 (October 2009), pp. 4258–4265.

[10] N. K. Jha, S. Ravi, A. Raghunathan, D. Arora, "Architectural support for safe software execution on embedded processors," *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, Seoul, Korea, 2006, pp. 106–111.

[11] B. W. Lampson, "Protection," *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437–443; reprinted in: *Operating Systems Review*, vol. 8, no. 1 (January 1974), pp. 18–24.

[12] N. Lin *et al.*, "Enforcing memory safety for sensor node programs," *Proceedings of the 12th IEEE International Conference on Computer and Information Technology*, Chengdu, China, October 2012, pp. 300–306.

[13] T. Liu, C. M. Sadler, P. Zhang, M. Martonosi, "Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet," *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, Boston, Massachusetts, USA, June 2004, pp. 256–269.

[14] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, W. Schröder-Preikschat, "Configurable memory protection by aspects," *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, Stevenson, Washington, USA, October 2007.

[15] L. Lopriore, "Access control mechanisms in a distributed, persistent memory system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 10 (October 2002), pp. 1066–1083.

[16] L. Lopriore, "Hardware/compiler memory protection in sensor nodes," *International Journal of Communications, Network and System Sciences*, vol. 1, no. 3 (August 2008), pp. 207–283.

[17] V. Potdar, A. Sharif, E. Chang, “Wireless sensor networks: a survey,” *Proceedings of the International Conference on Advanced Information Networking and Applications*, Bradford, United Kingdom, May 2009, pp. 636–641.

[18] T. Roosta, S. Shieh, S. Sastry, “Taxonomy of security attacks in sensor networks and countermeasures,” *Proceedings of the First IEEE International Conference on System Integration and Reliability Improvements*, Hanoi, Vietnam, December 2006, pp. 94–103.

[19] F. B. Schneider, “Least privilege and more,” *IEEE Security & Privacy*, vol. 1, no. 5 (September–October 2003), pp. 55–59.

[20] M. D. Schroeder, J. H. Saltzer, “A hardware architecture for implementing protection rings,” *Communications of the ACM*, vol. 15, no. 3 (March 1972), pp. 157–170.

[21] O. Stecklina, P. Langendörfer, H. Menzel, “Towards a secure address space separation for low power sensor nodes,” *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems*, Algarve, Portugal, March 2011.

[22] O. Stecklina, P. Langendörfer, H. Menzel, “Design of a tailor-made memory protection unit for low power microcontrollers,” *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems*, Porto, Portugal, June 2013.

[23] Y. Xiao *et al.*, “A survey of key management schemes in wireless sensor networks,” *Computer Communications*, vol. 30, no. 11–12 (September 2007), pp. 2314–2341.