

Aonix

The Ravenscar Tasking Profile for High Integrity Real-Time Programs

A Paper Presented at
Reliable Software Technologies -- Ada-Europe '98

Uppsala, Sweden, June 1998



The Ravenscar Tasking Profile for High Integrity Real-Time Programs

Alan Burns, Brian Dobbing, George Romanski

Abstract: *The Ravenscar Profile defines a simple subset of the tasking features of Ada in order to support efficient, high integrity applications that need to be analysed for their timing properties. This paper describes the Profile and gives the motivations for the features it does (and does not) include. An implementation of the Profile is then described in terms of development practice and requirements, run-time characteristics, certification, size, testing, and scheduling analysis. Support tools are discussed as are the means by which the timing characteristics of the run-time can be obtained. The important issue of enforcing the restrictions imposed by the Ravenscar Profile is also addressed.*

Introduction

High-integrity systems traditionally do not make use of high-level language features such as Ada tasking. This is despite the fact that such systems are inherently concurrent. Concurrency is viewed as a “systems” issue. It is visible during design and in the construction of the cyclic executive that implements the separate code fragments, but it is not addressed within the software production phases. Notwithstanding this approach, the existence of an extensive range of concurrency features within Ada does allow concurrency to be expressed at the language level with the resulting benefits of having a standard approach that can be analysed and checked by the compiler, and supported by other tools.

The requirement to analyse both the functional and temporal behaviour of high integrity systems imposes a number of restrictions on the concurrency model that can be employed. These restrictions then impact on the language features that are needed to support the model. Typical features of the concurrency model are as follows.

- A fixed number of activities (we shall use the Ada term *task* to denote an independent concurrent activity).
- Each task has a single invocation event, but has a potentially unbounded number of invocations. The invocation event can either be temporal (for a time-triggered task) or a signal from either another task or the environment. A high-integrity application may restrict itself to only time-triggered tasks.
- Tasks only interact via the use of shared data. Updates to any shared data must be atomic.

These constraints furnish a model that can be implemented using fixed priority scheduling (either preemptive or non-preemptive) and analysed in a number of ways:

- The functional behaviour of each task can be verified using the techniques appropriate for sequential code (e.g. [1]). Shared data is viewed as just environmental input when analysing a task. Timing analysis can ensure that such data is appropriately initialised and temporally valid.
- Following the assignment of temporal attributes to each task (period, deadline, priority, etc.), the system-wide timing behaviour can be verified using the standard techniques in fixed priority analysis (e.g. [2]).

Tasking Features

The Ada95 language revision has both increased the complexity of the tasking features and provided the means by which subsets (or profiles) of these features can be defined. To all of the Ada83 features (dynamic task creation, rendezvous, abort) has been added protected objects, ATC (asynchronous transfer of control), task attributes, finalization, requeue, dynamic priorities and various low-level synchronization mechanisms. Subsets are facilitated by pragma `Restrictions` that allows various aspects of the language to be limited in scope or removed from the programmer completely.

Whilst the full language produces an extensive collection of programming aids (e.g. see [3]) from which higher-level abstractions can be constructed, there are a number of motivations for defining restricted models:

- increase efficiency by removing features with high overheads
- reduce non-determinacy for safety-critical applications
- simplify run-time kernel for high-integrity applications
- remove features that lack a formal underpinning
- remove features that inhibit effective timing analysis

Of course, the necessary restrictions are not confined to the tasking model, but this paper only considers concurrency. To implement a restricted concurrency model in Ada requires only a small selection of the available tasking features. At the Eighth International Real-Time Ada Workshop (1997) the following profile (called the *Ravenscar Profile*) was defined for high-integrity, efficient, real-time systems [4].

The Ravenscar Profile

The Ravenscar Profile is defined by the following:

- *Task type and object declarations at the library level* -- that is, no hierarchy of tasks, and hence no exit protocols needed from blocks and subprograms.

- *No unchecked deallocation of protected and task objects* -- removes the need for dynamic objects.
- *No dynamic allocation of task or protected objects* -- removes the need for dynamic objects.
- *Tasks are assumed to be non-terminating* -- this is primarily because task termination is generally considered to be an error for a real-time program which is long-running and defines all of its tasks at startup.
- *Library level Protected objects with no entries* -- these provide atomic updates to shared data and can be implemented simply.
- *Library level Protected objects with a single entry* -- used for invocation signaling; but removes the overheads of a complicated exit protocol.
- *Barrier consisting of a single boolean variable* -- no side effects are possible and exit protocol becomes simple.
- *Only a single task may queue on an entry* -- hence no queue required; this is a static property that can easily be verified, or it can lead to a bounded error at runtime.
- *No requeue* -- leads to complicated protocols, significant overheads and is difficult to analyse (both functionally and temporally).
- *No Abort or ATC* -- these features leads to the greatest overhead in the run-time system due to the need to protect data structures against asynchronous task actions.
- *No use of the select statement* -- non-deterministic behaviour is difficult to analyse, moreover the existence of protected objects has diminished the importance of the select statement to the tasking model.
- *No use of task entries* -- not necessary to program systems that can be analysed; it follows that there is no need for the accept statement.
- *“Delay until” statement but no “delay” statement* -- the absolute form of delay is the correct one to use for constructing periodic tasks.
- *“Real-Time” package* -- to gain access to the real-time clock.
- *No Calendar package* -- “Real-Time” package is sufficient.
- *Atomic and Volatile pragmas* -- needed to enforce the correct use of shared data.
- *Count attribute (but not within entry barriers)* -- can be useful for some algorithms and has low overhead.
- *Ada.Task_Identification* -- can be useful for some algorithms and has low overhead, available in reduced form (no `Abort_Task` or task attribute functions `Callable` or `Terminated`)
- *Task discriminants* -- can be useful for some algorithms and has low overhead.

- *No user-defined task attributes* -- introduces a dynamic feature into the run-time that has complexity and overhead.
- *No use of dynamic priorities* -- ensures that the priority assigned at task creation is unchanged during the task's execution, except when the task is executing a protected operation.
- *Protected procedures as interrupt handlers* -- required if interrupts are to be handled.

The inclusion of protected entries allows event based scheduling to be used. For many high integrity systems only time triggered actions are employed, hence such entries and their associated interrupt handlers are not required.

The profile defines dispatching to be *FIFO within priority* with protected objects having *Ceiling Locking*. However it also allows a non-preemptive policy to be defined. Cooperative scheduling (that is, non-preemption between well-defined system calls such as “delay until” or the call of a protected object) can reduce the cost of testing as preemption can only occur at well-defined points in the code. It can also reduce the size of the run-time.

With either dispatching policy, the Ravenscar Profile can be supported by a relatively small run-time. It is reasonable to assume that a purpose-built run-time (supporting only the profile) would be efficient and “certifiable” (i.e. built with the evidence necessary for its use in a certified system). An equivalent run-time for a constrained Ada 83 tasking model has already been used in a certified application [5].

With the profile, each task should be structured as an infinite loop within which is a single invocation event. This is either a call to “delay until” (for a time triggered task) or a call to a protected entry (for an event triggered task).

The use of the Ravenscar profile allows timing analysis to be extended from just the prediction of the worst-case behaviour of an activity to an accurate estimate of the worst-case behaviour of the entire system. The computational model embodied by the Ravenscar profile is very simple and straightforward. It does not include, for example, the rendezvous or the abort, and hence does not allow control flow between tasks (other than by the release of a task for execution in the event triggered model). But it does enable interfaces between activities (tasks) to be checked by the compiler.

Preemptive execution, in general, leads to increased schedulability and hence is more efficient in the use of system’s resources (e.g. CPU time). As preemption can occur at any time, it is not feasible to test all possible preemption points. Rather, it is necessary for the run-time system (RTS) to guarantee that the functional behaviour of a task will not be affected by interrupts or preemption. For a high integrity application evidence to support this guarantee would need to be provided by the compiler vendor (or RTS supplier). For the Ravenscar profile the RTS will be simple and small.

Not only does the use of Ada increase the effectiveness of verification of the concurrency aspects of the application, it also facilitates a more flexible approach to the system’s timing requirements. The

commonly used cyclic executive approach imposes strict constraints on the range and granularity of periodic activities. The Ravenscar profile will support any range and a fine level of granularity. So, for example, tasks with periods of 50ms and 64ms can be supported together. Moreover, changes to the timing attributes of activities only requires a re-evaluation of the timing analysis. Cyclic executives are hard to maintain and changes can lead to complete reconstruction.

In a control system information may be translated through several stages. Input of sensor data may be scaled, filtered, used in control law calculations, scaled for output and finally output to a transducer. Safety critical standards e.g. DO-178B 6.4.4.2(a) requires that “*the analysis should confirm the data coupling and control coupling between the code components*”. This has been achieved in the past using cyclic executives which pass data between the code components in strict sequence. With the introduction of more sophisticated sensors, and the requirements to build more responsive systems, the data input and output rates and the rates of the computational processes may not be the same. A natural mapping for such systems is to use tasks with event triggers which enable data to be acquired, processed and output to transducers, at rates which are optimal for each processing step. Events provide a direct link between data and the code used in its processing. The Ravenscar profile facilitates the construction of concurrent programs where the code/data coupling is controlled, defined by the language and checked by the compiler (in contrast to facilities offered by run-time kernels defined independently of the language). The analysis to confirm coupling would be performed by code reviews to show that data is only accessed through synchronised or protected constructs.

Finally, note that the inclusion of a small number of event triggered activities does not fundamentally change the structure of the concurrent program or the timing analysis, but it does impose significant problems for the cyclic executive. Polling for ‘events’ is a common approach in high integrity systems; but if the ‘event’ is rare and the deadline for dealing with the event is short then the time triggered approach is very resource intensive. The event triggered approach will work with much less resources.

Implementing The Ravenscar Profile

Ada compiler vendor Aonix has undertaken the development of an Ada95 compilation system which implements the Ravenscar profile, known as *Raven* [6], hosted on Windows NT and Sparc Solaris, and targeting the PowerPC range of processors. This section describes some of the key elements of this implementation.

Development Practices

The principle goal of the implementation was to develop a runtime system for Ada95 restricted as per the Ravenscar profile, which was suitable for inclusion in:

- A safety-critical application requiring formal certification
- A high-integrity system requiring functional determinism and reliability
- A concurrent real-time system with timing deadlines requiring temporal determinism, e.g.

schedulability analysis

- A real-time system with execution time constraints requiring high performance
- A real-time system with memory constraints requiring small and deterministic memory usage

Consequently, a rigorous set of development practices was enforced based on the traditional software development model, including:

- Documentation of the software requirements
- Definition and documentation of the design to meet these requirements, including traceability
- Formal design reviews
- Formal code walk-throughs of the runtime implementation
- Definition and documentation of the runtime tests to verify correct implementation of the design
- Documentation of the formal verification test results
- Capture of all significant items within a configuration management system

Requirements

The software requirements include the following elements:

- The runtime design shall support both the preemptive and non-preemptive implementations of the Ravenscar profile.
- The runtime design shall optimise a purely sequential (non-tasking) program by not including any runtime overhead for tasking.
- The design shall structure the runtime such that a library of additional runtime Ada packages which have not undergone formal certification can be supplied as a stand-alone “extras”, for applications which require the extra functionality but not the rigors of certification.
- The runtime algorithms shall be coded such that the worst case execution time is deterministic and as short as possible.
- The runtime algorithms shall be coded such that the average case execution time is as short as possible.
- The runtime algorithms shall be coded so as to minimise the use of global data, and so as not to acquire memory dynamically. (The total global memory requirement of the runtime system shall be small and deterministic.)
- The runtime algorithms shall be coded so as to conform to the certification coding standards.
- The runtime algorithms shall be coded so as to conform to the Ravenscar profile plus sequential code restrictions.

- A coverage analysis tool shall be provided for certification purposes.
- A schedulability analyser shall be provided which supports standard algorithms used in fixed-priority timing analysis.
- Enforcement of the Ravenscar profile, plus other restrictions on sequential constructs, shall be performed at compile-time wherever possible. (This eliminates runtime code to perform the checks, and the risk of runtime exceptions being raised in the event of check failure.)
- The compilation system tools shall be verified using the Ada Compiler Validation Capability (ACVC) test suite, by coupling the tools to an alternate runtime system for the same target processor family, and which supports full Ada95. Runtime algorithms, which are common to the Raven runtime and the alternate runtime, shall also be verified in this way.
- The runtime kernel shall be verified using the verification tests written to validate the correct implementation of the requirements.

Design Considerations

Enforcing the Restrictions: The Ravenscar profile restrictions apply only to the concurrency model. It was therefore necessary first to define the additional restrictions that apply to sequential code. These are not defined in this paper, but in essence they follow the same goals of ensuring deterministic execution, simplifying the runtime support, and eliminating constructs with high overhead.

The requirement of enforcing as many restrictions as possible at compile time was met using the Ada95 pragma `Restrictions` [RM section 13.12]. A few of the needed restrictions were already defined using standard restriction identifiers in RM sections D.7 and H.4. However, most of the restrictions required new (implementation-defined) identifiers. It is expected that if the Ravenscar profile is included in the set of language features supported by the ISO Annex H Rapporteur Group, then a standard set of restriction identifiers covering the profile will be defined at that point in time. Within the Raven implementation, the set of needed restriction pragmas are supplied in source form to facilitate compilation into the Ada program library as configuration pragmas.

A compiler that enforces a subset to satisfy safety requirements needs to be carefully constructed. The compilation algorithms should not be changed to implement a particular subset, thereby preserving the value of its maturity and testing, including ACVC validation. This is an important means of raising the trust in the correctness of the toolset being used. Instead, the changes to generate the subset compiler are confined to reporting on violations of the subset in response to the presence of pragma `Restrictions`.

Two of the Ravenscar profile restrictions are enforced at runtime:

- Violation of `No_Task_Termination` is classed as a bounded error, which is defined to cause permanent suspension of the task. A mechanism to invoke a user-written handler for this situation is provided, which gives a hook for the application to apply remedial action.

- Violation of `Max_Entry_Queue_Depth=1` is a runtime check since the implementation has chosen not to restrict each protected entry to having only one statically determinable calling task, in keeping with the corresponding model which Ada95 uses for Suspension Objects [RM section D.10(10)]. Consequently violation of this restriction results in `Program_Error` exception being raised.

Runtime system code which processes bounded error conditions or raises exceptions when a restriction is violated, known as deactivated code (not dead code), is not excluded from certification considerations. DO-178B states [7] that the “software planning process should describe how the deactivated code will be defined, verified and handled to achieve system safety objectives.” [DO-178B section 4.2(h)]. Coverage testing of this deactivated code is also required by DO-178B: “... additional test cases and test procedures (should be) developed to satisfy the required coverage objectives.” [DO-178B section 6.4.4.3(h)]. Thus the level of trust of this error handling code is the same as that of the remainder of the runtime system.

Compilation Unit Closures: The requirements for there to be no runtime overhead due to tasking in a purely sequential (non-tasking) program, and that a non-certifiable library of packages be available stand-alone, providing Ada features beyond the basic kernel functionality, are met using coding conventions regarding closures of library units, as regulated by the use of Ada context (‘with’) clauses.

Three runtime unit closures are defined: for the sequential program kernel, for the tasking program kernel, and for the ‘extras’ packages. The coding standards are such that the sequential kernel units are not allowed to ‘with’ tasking kernel units, and neither the sequential nor the tasking kernel units are allowed to ‘with’ ‘extras’ units. Thus the separation of concerns (sequential versus tasking and certifiable versus not-certifiable) is enforced by the compiler using Ada semantics.

Other Runtime Constraints: The principle requirements governing the style of coding to be used for the runtime system are highly compatible and complementary, leading to algorithms which are small, easy to understand, and functionally and temporally deterministic, coupled with use of simple static data structures.

Certifiability: The requirements for certifiability impinge on the source code by means of specifying fixed format header comments for compilation units and all subprograms. The information in these headers includes:

- Overview of purpose or functionality
- Requirement(s) which are met
- Detailed definition of global data / parameter usage
- Detailed definition of algorithm

This description is checked against the actual code during walk-through audits, and is used to verify that the implementation conforms to the design, and that the design fully meets the requirements.

Performance: Several techniques are used to improve the performance of the runtime. Simple and very short runtime subprograms can be defined as having calling convention *Intrinsic* [RM section 6.3.1], which means that their code is built into the compiler and is used directly in place of the call. Typically this is used for immutable code sequences such as arithmetic and relational operators for types such as `Time` and `Time_Span` in package `Ada.Real_Time` [RM section D.8] and for highly time critical simple operations such as getting the identity of the currently-executing task.

Other short subprograms can be defined as being *inlined* [RM section 6.3.2], which gives similar performance gain by avoiding the procedure call and return overhead, but without having to actually build the generated assembler code into the compiler code generator.

In addition, since the runtime code itself must abide by the restricted Ada subset, this automatically excludes use of non-deterministic and dynamic constructs, plus those with high execution overhead or code size. Thus the code is written using simple Ada constructs which translate to equivalently simple assembler code, making it fast to execute, easy to verify, readable and maintainable.

Worst Case Execution Time: In order to perform accurate schedulability analysis, it is necessary to input the runtime execution overhead (see [8]). For hard real-time systems in which the failure to meet a hard timing deadline is catastrophic to the entire system, worst case execution times are generally used in the computations. The user can generally either analyse the Ada code [9] or measure the worst case time for application code using tests that exercise the various code paths, but for the runtime system operations, the user has no direct way of knowing which scenario will produce the worst case time, unless the runtime source code is available and also documentation to describe the criteria which determine the execution path at each decision point.

Thus, for every runtime operation with variable execution time, or whose operation can include a voluntary context switch, the vendor must provide metrics which typically define the worst case execution time either as an absolute number of clock cycles or as a formula based on application-specific data (e.g. number of tasks). For the runtime tasking kernel implementing the Ravenscar profile, this set of metrics will include:

- Entry and exit times for protected operations, including entry calls and barrier evaluation
- Entry and exit times for processing of the delay until statement
- Timer interrupt and user interrupt overheads
- Rescheduling times, such as the time to select a new task to run and the time to perform a context switch

Clearly, this imposes strict constraints on the algorithms used to implement these operations such that their worst case execution time is not overly excessive. For example, use of a linear search proportional to the maximum number of tasks in the program would be unacceptable for a program with a large number of tasks. So, the runtime contains optimisations to minimise critical worst case timings.

Runtime Size: The runtime was designed and coded to minimise the size of both the code and the data. For example, an important optimisation in the Ada pre-linker tool (the “binder”) is elimination of uncalled subprograms from the executable image. But this optimisation is only fully effective if the code is structured in a very modular way. For example, the runtime treatment of user-defined interrupt handlers as protected procedures should not be included in the image if interrupts are not used by the program. A more extreme example of this is the requirement that no code or data which is specific to the tasking kernel should be included in the image if the program does not use tasking.

In addition to this, the coding of the runtime data and algorithms was carefully crafted to optimise on speed and space, taking advantage of the various optimisations supported by the compiler.

Regarding data usage, the runtime does not make any use of dynamically-acquired memory, which is also a restriction on the sequential code of the application, thereby eliminating the need to support a heap with its associated non-determinism during allocation. The global data used is as small as possible, exploiting packing of data except where poor-quality code would be generated to access it. The data is packaged so that it is eliminated if the feature that it supports is not used (e.g. the interrupt handling table is eliminated when there are no interrupts in the program). The major component of the runtime data is the stack and Task Control Block (TCB) which is required for each task’s execution. Each application program is required to declare the memory areas to be used for the stacks and TCBs in the Board Support Package. This provides a simple interface to tune the stack sizes to the worst case values, whilst also giving full application-level determinism on the amount of storage which is reserved for this purpose.

Additional Supporting Tools

The additional tools which have been included in the implementation to support certification and schedulability analysis include:

- Condition code and Coverage Analysis tool
- Schedulability Analyser and Scheduler Simulation tool

Coverage Analysis (AdaCover): Under the DO-178B guidelines [7], it is necessary to perform coverage analysis to show that all the object code (both the application program part and the Ada runtime system) has been executed, including all possible outcomes of conditions, by the verification tests. The entire runtime system is subjected to coverage analysis as part of its auditing process. For the user application code, the tool AdaCover is provided to assist in formal certification.

AdaCover is in two logical parts:

- A target-resident monitor which records the execution of every instruction in the program, including the results of every decision point.

- A host-resident tool which annotates the compiler-generated assembly code listings with the results of stage 1, thereby providing the user with a report of coverage at either the object code or source code level, for the set of executed verification tests.

Schedulability Analysis (PerfoRMAx): The PerfoRMAx tool embodies classic schedulability analyser and scheduler simulation functionality. Given a definition of the actions performed by the tasks in the application in terms of their priority, execution time, period and interaction with shared resources, plus certain runtime system overhead times, the tool performs analysis of the schedulability of the task set based on a user-selectable scheduling theory, for example Rate Monotonic Analysis (RMA) [2].

The tool is also able to provide a graphical view of the processor load based on a static simulation of the scheduling of the tasks by the runtime system, thereby giving clear indication of potential regions of unschedulability. If such regions exist, the tool outputs messages highlighting the cause of the unschedulability together with suggestions for corrective action.

Testing

The testing activity is split into two components:

- Use of the ACVC test suite to verify the validation status of the compiler, binder and code generator support routines
- Development of a specific test suite to certification level for the Raven runtime.

ACVC Testing: Since a substantial number of ACVC tests violate Ravenscar profile restrictions, particularly relating to the tasking tests, it is not possible under current rules to validate such a subset. However by use of the same compilation system tools linked to a full Ada95 runtime system for the same target processor family, it is possible to run the full ACVC suite, thereby validating the correctness of the compiler, binder and common code generator support routines (e.g. block move). The validated compiler contains all the processing to treat pragma Restrictions, but since the tests do not include these pragmas in the source code, no enforcement of the subset is performed and hence all the tests can execute.

Certification Tests: A test suite has been created to verify the correctness of the kernel runtime subprograms, thereby complementing the ACVC testing (which was not able to test these), whilst also ensuring the level of reliability specified by the requirements.

Each test contains a header in the source code that includes:

- Identification of the requirement to be tested
- Identification of the runtime module under test
- Test description
- Test case definition, including inputs and expected results

The results of executing the tests against each baseline development of the runtime system are documented.

Packaging

When the Raven product is purchased, an option is available to purchase separately all the material required for formal certification. This option includes:

- Full runtime source code
- Full development documentation which is relevant to certification
- Full test pack, including sources, scripts and documentation, so that the tests can be re-executed on the runtime code during formal certification of an application.

Conclusion

This paper has described the Ravenscar profile, a subset of Ada95 tasking intended to model concurrency in safety-critical, high-integrity, and general real-time systems. The use of a powerful, structured and highly-checked language such as Ada is vitally important in all market sectors demanding high reliability and efficiency.

The paper has also described a commercial-off-the-shelf implementation of the profile for the PowerPC processor family which has proved the feasibility of developing production-quality tool support and a certification-quality runtime system for the Ravenscar profile.

On-going work within the International Standards Organisation Working Group 9 exists to incorporate the profile concepts within the recommendations on the use of Ada in high integrity systems.

References

- 1 *High Integrity Ada – The SPARK Examiner Approach*, J. Barnes, Addison Wesley Longman Ltd (1997)
- 2 *A Practioner’s Handbook for Real-Time Analysis : A Guide to Rate Monotonic Analysis for Real-Time Systems*, M.H.Klein et al, Kluwer Academic Publishers (1993)
- 3 *Concurrency in Ada*, A. Burns and A.J.Wellings, Cambridge University Press (1995)
- 4 *Proceedings of the 8th International Real-Time Ada Workshop: Tasking Profiles* ACM Ada Letters (September 1997)
- 5 *T-SMART – Task-Safe Minimal Ada Real-time Toolset*, B. Dobbing and M. Richard-Foy in *Proceedings of the 8th International Real-Time Ada Workshop*, pages 45-50 ACM Ada Letters (September 1997)
- 6 *ObjectAda/Raven Compilation System for PowerPC*, Aonix (1998)
- 7 *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B/ED-12B. RTCA Inc (December 1992)
- 8 *Engineering and Analysis of Fixed Priority Schedulers*, D.Katcher et al. In *IEEE Trans. Software Engineering* 19 (1993)
- 9 *Combining Static Worst-Case Timing Analysis and Program Proof*, R.Chapman, A. Burns, A.J.Wellings. In *Real-Time Systems* 11(2):145-171 (September 1996)
- RM *Ada95 Reference Manual*, ANSI/ISO/IEC-8652:1995, Intermetrics Inc. (January 1995)

To obtain more information, please contact Aonix at www.aonix.com or your local Aonix office.

Aonix World Headquarters
Phone: (800) 97-AONIX
Fax: (619) 824-0212
E-mail: info@aonix.com



France
Phone: +33 (0) 1 41 48 11 00
Fax: +33 (0) 1 41 48 10 20
E-mail: info@fr.aonix.com

United Kingdom
Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 571866
E-Mail: info@aonix.com.uk

Karlsruhe, Germany
Phone: +49 (0) 7 21/9 86 53-0
Fax: +49 (0) 7 21/9 86 53-98
E-mail: info@aonix.de

München, Germany
Phone: +49 (0) 89/45 10 57-0
Fax: +49 (0) 89/45 10 57-30
E-mail: info@aonix.de

Sweden
Phone: +46 (0) 8 601 9491
Fax: +46 (0) 8 601 9495
E-mail: info@aonix.se