Alphorn: A Remote Procedure Call Environment for Fault-Tolerant, Heterogeneous, Distributed Systems

Alphorn (pronounced Alp-horn) is a software environment for programming distributed computer systems. Programs running on different computers, possibly of different types and running different operating systems, communicate in a client-server relationship by means of remote procedure calls. This efficient construct structures programs neatly and relieves programmers of caring about where their modules will run.

Distributed computer systems like those used for industrial process control are large and complex. Although connecting computers in a local area network is relatively easy, programming them requires skilled personnel, especially when the project becomes large and must be extended at runtime. The tools developed by mainframe manufacturers for single-processor machines fall short in distributed systems. In particular, they do not meet the important requirements of industrial control systems: distribution, real-time response, and fault tolerance.

Development environments that support the whole software cycle, integrating programming, distribution, configuration, communication, and debugging, are appearing. They try to relieve the programmer of dealing explicitly with distribution, real-time behavior, or replication. Alphorn, a programming environment developed by our group, goes further in that direction: an application consists of a collection of modules, programmed in standard Modula-2, for instance.2 These modules are written as if they were to be linked to form a single task running on a single processor. Indeed, that is the case when no distribution is required.

All communication between modules takes place through procedure calls, an approach that gives programmers a familiar view of the system. Modules interact on a client-server basis. A module exports a number of procedures, which are the services that client modules will call. The internal data structures of the modules are not remotely accessible. Programmers must obey the rules of data encapsulation.

In a distributed system, the server and client modules run on different computers, different processors of the same computer, or different tasks of the same processor. Local procedure calls be-
come remote procedure calls, but for the programmer, little is changed. Copies of a module may be loaded on different nodes to provide several instances of a service—for instance, to support several printers or to provide fault tolerance. An automatic generator makes the services of a local module remotely accessible through a stub mechanism. To this effect, the services of a module are expressed in a service interface language, which is an extension of Modula-2.

The distribution of modules on the processors and tasks is decided only at configuration time, when the work load and location of the processors are known. An interactive configurator generates the “bug” for the modules and builds the tasks. Each task includes runtime support that starts the clients, adapts the data presentation, and controls communication. The data travel over the available media: common memory, local area network, or open network. The remote procedure call (RPC) protocols are used all along, even for communication at the lowest level.

Alphorn has been ported from PDP 11/RSX to VAX/VMS, Sun/Unix, VME/VersaDOS, IBM PC, Honeywell/GCOS, and to specialized programmable logic controllers. Although these machines use different processors, languages, and compilers, they communicate transparently with the same mechanism. Alphorn has proved useful as a structuring tool for large applications as well as a communication tool over a variety of networks.

**Distributed process control**

A typical distributed control system consists of computer nodes interconnected by a local area network such as Ethernet or token bus (IEEE 802.4). The nodes may be of different types and run different operating systems. They may themselves consist of multiple processors interconnected by a parallel bus and sharing a common memory. In process control, the computers are assigned dedicated tasks, as Figure 1 (on the next page) suggests.

The communication takes different paths, depending on the location of the partners:

- **Nodes** communicate through the network by sending and receiving messages. They do not share a common address space. A name server may provide a systemwide directory service.

- **Processors** within a multiprocessor communicate through a common memory, which may be located globally on the parallel bus or may consist of dual-ported local memories. The processors may take advantage of the message-passing facility offered by some parallel buses, such as Multibus II. They may form a pool or be of different types.

- **Tasks** executed by the same processor share the same physical memory but are prevented from accessing each other’s region by the memory management unit. They communicate through shared-memory regions using services of the kernel. (Tasks are called processes in VAX/VMS or Unix terms.)

- **Threads** are parallel activities within the same task. Threads (called processes in Modula-2 and light-weight processes in Unix) are based on coroutines. They share the address space of their task and communicate with
Figure 1. A distributed process control system.

Each other through common variables. Threads of the same task cannot interrupt (preempt) each other, but a thread runs until it reaches a synchronization point. The task holds a small scheduler that determines the next thread to run. Because of its synchronous nature, a thread switch is faster than a task switch. (Threads are the smallest unit of parallelism we consider here.)

The communication between these entities is illustrated in Figure 2. Since communication paths vary widely, we wish to offer the programmer a unified communication interface independent of the communication path.

Traditionally, communication between remote entities has been based on messages. Logical channels, which implement point-to-point or multipoint links over the same physical medium, carry the messages. Channels are called mailboxes, pipes, ports, and so on, with subtle variations in meaning. The programmer first opens a channel and then sends data explicitly by using something like a Send Message operation. The partner receives the message by a Receive Message operation. These operations can be inserted at any place in a program, allowing great freedom but also great responsibility to the programmer. Message passing lends itself to spaghetti-code programming, which is hard to debug and maintain, especially when several entities write and read to the same channel.
While the communication system cannot avoid some kind of message passing, the application programs should be better structured. Alphorn relies on the Remote Procedure Call (RPC) as an alternative to message passing. The difference between messages and RPCs is comparable to that between Fortran and structured languages without Goto statements. Programming with RPC considerably influences programming style. When RPCs are properly used, there is no need for other communication constructs.

**Programming style**

Programming in Alphorn is based on the concept of objects, which is supported by high-level languages. An object consists of data structures and a well-defined set of procedures giving access to the data. An example of such an object is the Oracle database; the contents of the database can be accessed only through procedures such as GetItem, InsertItem, RemoveItem, and SearchItem. These procedures form the object interface, which is all a user must know to access the database. The database can be implemented by arrays, lists, records, and so on, which the user need not know about.

Modula-2 expresses the object interface in the definition module, which lists all exported variables, data types, and procedures. The implementation of the object is contained in the implementation module, which is a private affair of the object. The database user ignores how and where the items are stored.

A module that exports procedures becomes a server when called by a client program. An example of a service call appears in Figure 3. The server “Bank” is defined by a server module. Its server interface is defined in the definition file “Bank.def.” The procedures exported by the definition module (for example, GetItem) are the services of that server module. The services are programmed in the implementation file “Bank.mod.”

The calling conventions of the client

continued on p. 60
must match the exported definitions of the server module. Figure 4 shows a client that calls a server procedure when both modules are directly linked. The shaded key form symbolizes the interface.

**Remote procedure calls**

A remote procedure call looks exactly like a local procedure call to another module of the same task. For instance:

```plaintext
Bank.GetItem ("SMITH", Salary);
```

This call is issued by a client program to a server procedure Bank.GetItem. The client is suspended until the call returns. That is exactly how the familiar local procedure call works. Figure 5 shows a remote procedure call as a function of time.

In a local call, the called procedure belongs to another module of the same task. Communication takes place directly through procedure call mechanisms the compiler provides. This is the view of communication the programmer ideally should have.

In a remote call, the server belongs to a different task of the same or a different processor. For the client, however, everything should happen as if the call were local. In reality, the call is forwarded over the communication path. Bank.GetItem is executed by another task in the same computer (internal call) or in another computer (external call), which can be a different type of machine or can be running a different operating system. The results are returned in another message, as shown in Figure 6.

Remote procedure calls are very similar to local procedure calls. For instance, remote calls may be nested at will. But there are some differences:

- **Local variables.** Although most languages permit a module to export local variables, this is not allowed if the module is to be distributed; client and server residing on different sites share no common address space. Therefore, no reference to the address space of either the client or the server (common variables) may be used. This rule is in accordance with object-oriented programming. All information is transmitted in the parameters of the called procedure. In remote calls, parameters are always transmitted by value (even if the programmer passes parameters by reference).

- **Parallelism.** In contrast to a local procedure call, the same service may be called again by another client while a remote procedure call is executing. For each client, a different instance of the service is started. Each service is executed by a different thread. When the threads are exhausted, the calls are queued. The threads need synchronization among themselves to access common resources.

- **Blocking calls.** Although client and server may be executed by independent processors, they do not execute in parallel; the client remains blocked until the call returns, according to the semantics of a procedure call. A client may wait for the result of only one call at a time. This causes no loss of parallelism, however. The processor that was executing a blocked client is free to execute another service, the same service again (on behalf of another client), or another client until the call returns. When a call is blocked, another activity can start or resume execution.

- **Nonblocking calls.** In addition to blocking calls, a client may start multiple calls, provided these calls do not return results. Such calls are called remote procedure invocations. Unlike RPCs, RPIs require an underlying flow control mechanism, invisible to the user. A client may not wait for the completion of an RPI. Unlike Delta-4, Alphorn supports no asynchronous call with results (Request Service Request/Remote Service Wait), because this construct loopholes the structure. (In ANSA, blocking and nonblocking calls are called synchronous and asynchronous.)

- **Events.** Normally, a remote call is a one-to-one relation. Alphorn has no broadcast calls but can publish an event from a signaler to subscribed handlers. Programmers insisted on having this construct, but it does not support...
fault tolerance and we therefore do not encourage its use.

**Stubs**

The entity in charge of relaying the call is called a stub (see Figure 6). The client stub plays the role of a server for the client. The client accesses the client stub in the same way it would access a local service procedure. The client stub converts the call into a message, which is sent over the network or over shared memory to the destination site. There, it is received by the server stub, which plays the role of a client for the server on that site. The server executes the service and returns to the server stub, which forwards the result to the client stub. The client stub then returns the results in the same way a local service procedure would do.

Some languages, such as Argus,7 provide remote procedure calls as programming constructs. Most other languages cannot express that a procedure is called remotely or declare a module as server. Nor is this necessary, since we want to handle local calls and remote calls identically. To preserve the same interface for remote procedure calls as for local calls, we use stubs, not only for communication, but also as interfacing tools. The client stub presents the same interface to the client as the actual server would. In fact, the client stub simulates the existence of a local server. The server stub simulates a local client call to the actual server. The call interface is the same as it would be if the server were called locally. This is illustrated in Figure 6.

Stubs do much more than just transform a local call into a remote call. In fact, they are the centerpiece of the communication process. Stubs initialize and administer communication, control the success of calls, publish services over the network, and provide the hooks for debugging tools. In fault-tolerant systems, stubs control the correct replication and synchronization of redundant processors.2 In heterogeneous systems, stubs control the transformation of data formats.11,12

**Automatic stub generation.** The client and server stubs are normal modules, linked to the client and server respectively and forming part of their task. They could be written by hand. However, it is much more convenient and safer to generate them automatically with a stub generator.10 The RPCGEN stub generator analyzes the file that defines the server module. This server definition file is, of course, common to the client and server modules. RPCGEN generates the implementation modules of both the client and server stubs. The principle of stub generation is shown in Figure 7.

Each client receives a copy of the client stub. The stub generator also examines the symbol file generated by the compiler to detect version conflicts. The symbol file contains a version key, which the stub transmits with each message to enforce type checking over the network.

**Service interface language**

All RPC systems rely on a server definition file. It is called the network interface definition file in Apollo's NCS and the U/F specification in Delta-4.

The server definition file could be written in a standard computer language, like a definition file in Modula-2. Although simple applications could run with some default settings, this solution is not general enough. The stub generator needs additional informa-
tion which is not relevant when a procedure is called directly from within the same task. For instance:

- **Procedure type.** Remote procedure calls (blocking) and remote procedure invocations (nonblocking) are distinguished by an "RPC" or "RII" prefix.
- **Procedure parameters.** Since clients and servers do not share a common address space, the stubs copy all parameters, even those passed by reference (pointer). To avoid unnecessary copying, we distinguish parameters that the procedure may only read (input parameters, or "arguments"), parameters the procedure may modify (output parameters, or "results"), and parameters the procedure both reads and writes (input/output parameters). Ada is one of the few languages that tags the parameters as IN, OUT, or INOUT. We use the same convention in Alphorn.
- **Dynamic-size parameters.** The size of this type of parameters is not part of the interface, since it is known only at runtime. A dynamic array declaration allows us to copy these parameters correctly.
- **Parallelism.** The degree of parallelism (that is, the number of server threads) and the degree of fault tolerance (the number of replicates) can be indicated in the server definition file. This data can be modified at configuration time.
- **Presentation.** A description of the target machine is included in the interface definition so that the stubs can convert types automatically. Since the data type is part of the server interface, the client must adapt. For instance, a client stub converts 32-bit integers returned by a little-endian (Intel) server to its own big-endian (Motorola) format. Since the parties agree on the data representation at compile time, there is no need for an on-line negotiation of data types. This avoids a presentation layer in the communication process. A unified data type presentation can also be specified, similar to Sun's XDR.¹
- **Initialization.** Some clients come into existence when the system is started. These "top actions" are specified as services called from nowhere.

The server interface is expressed in a service interface language. ECMA, ROSE (Remote Operations Service Elements, ISO 9072), and Delta-4 use the standardized Abstract Syntax Notation 1 (ASN1, ISO 8824) to achieve independence of any computer language. However, ANSA and IEC SC21 consider ASN1 suitable for expressing message structures but too weak as an interface description language; they have developed their own interface description language, derived from Xerox's Courier protocol.¹¹

For its service interface language, Alphorn uses the syntax of a normal Modula-2 definition file, in which the stub control instructions are enclosed in comments. This allows features not found in ASN1. The main advantage is that the Modula-2 compiler can process the file directly. This is all that is needed to run with a single task. To make a module remotely accessible, the stub generator analyzes the stub control information enclosed in the comments and the symbol file (to retrieve the version key).

The use of Modula-2 has an interesting by-product. Modula-2, like most languages, was created as a sequential language and not to support parallelism. As such, it defines a static structure. Clients and servers, however, are dynamic entities. The Modula-2 syntax is used here to express a dynamic structure, which becomes apparent only at runtime. This interpretation is done by the stub generator.

**Configuration**

Once the servers, the top client modules, and the server stubs have been compiled, they are configured: that is, the entities belonging to the same site are linked together to form one runnable task. There may be several tasks in the same node, each one comprising servers and top modules. Server modules are not active by themselves; they need a client to call them. Top actions become active when the task is started. When they call a service, they become top clients. A task is formed by a main Modula program that directly imports the top modules and the server stub modules (Figure 8). The top modules import client stubs as needed, while the server stub modules import their corresponding server modules.

The main program provides the Runtime Support (RTS is similar to Deltase in the Delta-4 project and to the ANSA nucleus.) It imports both the top clients and the server stubs. The main program creates and manages a number of threads. Conceptually, all these threads are parallel—that is, they would execute in parallel if there were a sufficient number of processors. They form a pool that runs the servers and top actions. The runtime support provides the communication interface.

The writing of the main program is automated by the OBJGEN utility. The OBJGEN configurator generates a main program, which is a normal Modula-2 program, to hold the top client, client stubs, server stubs, and server modules. OBJGEN interprets the definition files as follows:

- For each procedure (without parameter) exported by a top module, a thread is created and started. These threads form the top actions. Conceptually, all top actions are started simultaneously. A top action becomes a top client when making a remote call.
- For each server stub, several server threads, or servers for short, are created. When an action calls a service, one of these servers will execute the corresponding procedure as a server action. The number of server threads can be specified in the definition file during stub generation. The default value is 3 in the present implementation.
### Runtime and debugging tools

A running system consists of a number of nodes, each running several application and system tasks. Network communication is only partially realized in the stubs. Other entities must exist to support them, in particular the networker. The network configuration is shown in Figure 9 on the next page.

Each task may contain servers and top actions. Top actions usually call services of other servers, but they need not. For instance, in Figure 9, Task 2, the top action of node Alpha, needs no access to a stub since it performs no remote calls.

**Networker.** A stub performs only part of the communication work. Basically, it provides the glue between the application interfaces and the standard low-level RPC mechanism. A driver called the networker executes the actual communication. The networker is permanently active; that is, it is a separate task or a device driver (except on single-task systems like MS-DOS, where it is linked to the application). This driver is common to all applications in a node, and it hides the details of communication from the stubs.

The networker executes the RPC protocols. It analyzes the available communication links—for instance, the shared-memory communication module, the Ethernet driver, DECnet to access another computer, or X.25. When the networker recognizes that client and server are in the same node, it connects the stubs directly through shared memory. Similarly, when a DECnet link exists between client and server, the networker uses its facilities directly.

**Name server.** When a client calls a server in the network, the networker must know the location of this server. That could be specified during configuration by early binding. Alphorn includes and removes separately compiled modules as the system runs. It uses late binding to indicate the location of the servers at runtime. Each networker performs the name server function—there is no central trailer. The name server publishes all services in the network. (It does not publish their interfaces, however, unlike the Delta-4 trailer. That would have required each node to have mass storage.) A multicast protocol continually updates all name servers.

The same service may be installed several times in the system, to increase performance or fault tolerance. In the first case, the user may wish to choose and distinguish which server is processing the call. This is especially important for exception handling. Thus, the name server provides a mechanism to identify the client and the server that participate in a call. When a service is replicated for fault tolerance, both the primary and the backup carry the same identification. The networker monitors the start and termination of server and clients, notifying all interested applications.

**NetMonitor.** During development and test, it is useful to have an insight on the way communication takes place and which entities are active at a time. The NetMonitor utility runs as a task anywhere in the network. It monitors existing sites, traces established connections, lists clients, servers, handlers, and signalers, and displays statistical information. Finally, NetMonitor can stop sites in an orderly way.

### Fault tolerance support

There is no general-purpose fault-tolerant computer. To handle redundancy effectively, a fault-tolerant system must take account of the plant or other process control application. Fault tolerance therefore requires close collaboration between the application programmer and the system designer. Alphorn provides the following basic tools to help them construct fault-tolerant systems.

**Exception handling.** As is not the case with a local call, one must be prepared for the failure of a remote call. To maintain the same interface as for local calls, a call's status is delivered by a "hidden" module called RPCStatus. Since the client is blocked during the call, a failure of the server must be signaled by an exception handler. This is a critical component; since some machines and languages support exception handling well, such as VAX/VMS, while others seem to ignore the problem. Much of the porting work for Alphorn consisted of implementing exception handlers for machines that lacked them.
**Error reporting.** Each node constantly monitors the network. Healthy nodes regularly communicate or send "I'm alive" messages; the absence of messages triggers reconfiguration. Therefore, the worst-case error detection latency is equal to the network's cycle time. This mechanism is part of the basic runtime support.

**Redundant distributed system.** In process control at the supervisory level, the main requirement is high availability. Basic availability is best provided by a duplex structure. Functional redundancy is provided by additional nodes in the network and duplication of communication links. When a node fails, its backup node takes over its function. The backup
node is functionally redundant to the on-line node. In particular, the backup has the same access to I/O devices as the on-line node, either through dual-ported devices or redundant devices.

Not all nodes need to be duplicated. The Alphorn fault tolerance concept supports both replicated and nonreplicated nodes in the same system. Higher levels of replication such as triplication (two backups) may also be integrated. Most fault-tolerant applications in Alphorn, however, are expected to be of the duplex type.

**Error detection.** The failure of a node is detected by means such as parity checks, self-checking circuits, watchdog timers, and the like. We assume that each node is fail-stop—it will stop sending data in case of failure (‘fail-silent’ in Delta-4). For process control at the supervisory level, integrity is not critical and the self-checking provided by commercial mainframes is generally sufficient. Full integrity—no false data in case of failure—is mandatory in critical applications such as railway signaling. In such cases, each node must be duplicated to provide sufficient coverage.

It makes no sense to increase fault detection coverage of hardware past a certain point. Software causes many errors, which can hardly be caught by hardware means. That is why Alphorn provides no protection against malicious behavior or bumbling nodes. We do not try to provide tolerance of software errors. However, it is important to report and treat software errors, even in nonredundant cases, so that the failure of one unit will not cause others to crash. For this purpose, Alphorn relies on its exception-handling mechanism.

**Redundancy actualization.** It is not enough to add redundant nodes to a network. Unless the state of backup nodes is close enough to that of their on-line units, switchover will be rough or even impossible. There are two basic techniques to keep backup computers actualized: standby, or periodical update, and workby, or parallel operation (called passive and active replicates in Delta-4; coordinated and parallel replica in ANSA).  

- In the standby (or asynchronous) mode, a backup node is regularly actualized by transfers from the on-line node. These state transfers take place at checkpoints. Otherwise, the backup is free to perform other tasks.
- In the workby (or synchronous) mode, the on-line node and the backup node perform the same tasks at the same time. Therefore, their internal states remain identical. In principle, their outputs could be compared to detect errors. The backup cannot be used for other tasks. Workby requires synchronization and matching to remove all sources of nondeterminism, which could let the units diverge. Synchronization and matching over the network costs time.

Alphorn supports both operation modes, but unlike ANSA and Delta-4, Alphorn supports no replica groups. In either mode, the backup must constantly monitor the network activity to reproduce or coexecute the actions of the primary. In both cases, a teaching mechanism updates a fresh node to backup status.

**Resilient remote procedure calls.** Standard RPCs can be easily extended to support both the standby and the workby modes of operation. The basic idea is that actualization of the backup, whether through checkpointing or synchronization, requires that communication over the network be monitored. To this effect, both the on-line nodes and their backups receive call and return messages (Figure 10).

For instance, when a call is made, it is received by the server and by the backup server. If the server fails, the backup will have received the service call and can execute it. Or, if the client fails, its backup can receive the return message in its place and proceed. Thus, each RPC becomes an implicit checkpoint. This method effectively eliminates the domino effect.

**Causal broadcast.** Actualization and synchronization of replicates over the network require reliable multicast communication. (Although some local networks provide acknowledged broadcast transmission, this property cannot be expected of open networks.) Rather than a full-fledged atomic broadcast protocol, Alphorn relies on a causal broadcast protocol. A causal broadcast protocol ensures that message order is maintained under all circumstances for all interested parties. This protocol is built on top of existing protocols such as...
Swiss PTT uses Alphorn for a distributed database of 200 nodes scattered over the country.

DEGnet or X.25. Gausal broadcast has proved to be very efficient and handy, even in nonreplicated applications. (Unlike Delta-4, this protocol does not provide authentication or signatures; protection against intrusion is unnecessary in a dedicated system.)

Application dependence. Ideally, the application programmer writes programs without caring about replication. It is indeed desirable to offer, for instance, a control system in a redundant and in a nonredundant configuration. Therefore, the code running in the on-line unit and in the backup unit must be identical. Redundancy is introduced at configuration time and hidden in the runtime system or in the device drivers. This ideal cannot be completely maintained.

Fault tolerance constructs. Alphorn provides useful constructs to support fault tolerance. While each RPC provides a synchronization/rollback point, it may be necessary to include checkpoints at closer intervals, especially for top actions, which are never called as server. To this end, Alphorn provides an iteration construct, which inserts a checkpoint each time a loop is executed.

In standby mode, data modified in the primary node are transmitted to the backup unit. Indiscriminate checkpointing, as used in transaction computers, is not acceptable in real-time systems. Alphorn provides a construct for tagging data structures that are subject to modification. Only these data will be transmitted to the standby when the call returns.

This checkpointing allows us to build atomic actions, which are either completely executed or not at all. When a failure occurs, computation starts from the last checkpoint (rollback) and proceeds to repeat operations already done by the failed unit (rollahead), until reaching the failure point. The rollahead procedure avoids repeating output operations already performed by the failed unit. But in some situations, it may be necessary to redo such operations—for example, refreshing the operator's screen. Here, the application dependency must be hidden in the I/O drivers. Alphorn does not, however, provide atomic transactions; this is considered an application issue.

Teaching backups. The teaching of new backups—for instance, after repair—is performed in the background and consists of a dialogue-like communication between an online task and the future standby task, to transmit the state of an object. Depending on the amount of spare computing time, two options are available: concurrent teaching goes on during normal operation but increases load. Off-line teaching requires a small interruption but costs no runtime overhead.

The original goal of the Alphorn project was to develop a fault-tolerant computer network. Later, the software environment and the network-independent communication based on RPC became the mainstream of the project. Although based on Modula-2, Alphorn is not limited to that language. Stubs may be easily generated for other languages, such as C. In fact, the target language, machine, and compiler are options of the stub generator. A Pascal version has been developed for a machine that has no Modula-2 compiler. In collaboration with the Swiss Federal Institute of Technology, in Lausanne, we are working to provide manufacturing message services (ISO 9506) as Alphorn services. Alphorn is now used for a distributed database implemented on behalf of the Swiss PTT (post office), consisting of 200 nodes scattered over Switzerland. That is in keeping with the project’s name: alphorns are traditional Swiss musical instruments derived from the long horns shepherds once used to call their herds across the valleys.

References


Hans-Ruedi Aschmann is a project leader in data communications with Glance Ltd. in Steimaur, Switzerland. He received the BS degree in electronics and the PhD degree from the Swiss Federal Institute of Technology in Zurich after postgraduate studies at Rensselaer Polytechnic Institute in Troy, New York.

Elisabeth Hoepli works as an informatic engineer for the Asea Brown Boveri Research Center in Baden. Her technical interests include real-time software and data integrity in communication systems. Hoepli received her BS degree in physics from the Swiss Federal Institute of Technology.

Niklaus Giger works as a consultant for the Asea Brown Boveri Research Center in Baden. He received his BS degree in electronics.

Peter Janak, a software engineer, leads the Non-Stop Laboratory at the Cantonal Hospital of Saint Gall, Switzerland, for ABB Corporate Research. After his studies at the Interstate College of Buchs, Switzerland, he pursued postgraduate studies at the Swiss Software School in Bern.

Hubert Kirrman leads the Computer Architecture Group at the Asea Brown Boveri Research Center. Previously, he taught courses in automatic control at the District University of Bogota, Colombia, and developed electrical sensors. Kirrman received his BS degree in electronics and his PhD degree from the Swiss Federal Institute of Technology. He is a member of the editorial board of IEEE Micro.

**Reader Interest Survey**

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

<table>
<thead>
<tr>
<th>Low 165</th>
<th>Medium 166</th>
<th>High 167</th>
</tr>
</thead>
</table>

October 1991  67