Interaction Support in a Kernel for the Embedded Environment

C Gray Girling

AT&T Research Cambridge
24a Trumpington Street
Cambridge, UK

Abstract

As any other systems, those produced for an embedded environment are better developed when specified and implemented in a modular fashion. This paper outlines some infrastructural abstractions that allow the interaction of a wide range of system components; and goes on to describe their implementation optimised for the simplicity of typical embedded applications in a kernel for a component-based operating system. At the same time it explores the extent to which these abstractions can apply to a standard model of "computational" elements in the Open Distributed Processing environment. It is argued that designing code to use the proposed abstractions may help to extend its utility beyond application to specific embedded systems. Finally, some examples of the practical use of such an infrastructure are described.

1 Introduction

The embedded system kernel discussed in this paper (the Embedded Environment Kernel, EEK) was developed originally in order to support a number of practical research projects within the laboratory but is now principally associated with the "Piconet" low-power room-area radio network project [Bennett et al 97]. Its lineage can be traced back to the "Tripos" operating system, developed in Cambridge University's Computer Laboratory in the late 70s and used in the Cambridge Distributed Computing System [Needham et al 82], which gave rise to a number of commercial and research operating systems including AmigaOS [Pountain 89], used in the Amiga personal computer, and ATMos [Virata 99], EEK's immediate precursor. Although these generations have shared almost no code with their successors they do share a number of design elements:

(1) pre-emptively scheduled light-weight processes;

(2) single address space in a single memory domain; and,

(3) cheap call-by-reference message passing.

Such kernels (particularly due to the use of a single memory domain) are appropriate only for a restricted range of applications. The lack of protection means that systems can incorporate only applications for which there is some reason to believe in their integrity, and also means that systems are unlikely to scale well. The latter is due to the inverse relationship between integrity and system complexity. None-the-less there is a class of systems, with a small and temporally fixed number of parts that can take advantage of the higher performance that call-by-reference messaging delivers. These generally fall under the heading of "embedded systems". It is this type of system for which EEK is primarily intended, although it is also appropriate in hardware platforms that cannot support memory protection.[1]

However, it is a matter of everyday observation that embedded systems, or at least their software components, very often do continue to evolve. Eventually increasing complexity may mean that re-implementation in a different environment (for example one with separate memory domains or one in which the application is distributed between machines) becomes the only practical way forward. If interaction between system components (e.g. message passing) has not been handled according to good design principles this form of portability will be difficult or impossible to achieve.

[1] When address spaces are very large (e.g. 64 or 124 bit) a measure of security can be achieved through the capability-like nature of an address – this type of operating system may also be relevant here.

It is for this reason that one of EEK's design goals was to use intercommunication primitives that would be appropriate for the general case but which would have the cheapest possible implementation in an embedded system.

2 Message Interfaces

It was once said that, no matter what the operating system, message passing back and forth between protected domains would run no faster than 50Hz. Since then the search for faster ways of interacting has developed in two main directions: exploring the alternative of messageless monitor-only systems (e.g. [Lauer et al 78]); and, designing systems that employed some form of lighter-weight process. Whilst there is still informed debate as to whether a message-based or procedure-call-based interaction paradigm is "best" the former is at least as respectable as the latter, being used both in communication formalisms, such as CSP [Hoare 85], and in standard models, such as International Standards Organization's (ISO's) ODP [ISO 95]. Today, particularly in the domain of distributed processing, code is structured in terms of the "process", "task", "light-weight thread", "coroutine", or "activation" in preference to re-entrant guarded objects and monitors.

In addition to the specific communication mechanisms supported by EEK (call-by-reference messages) a generic messaging environment must apply to a variety of other implementations. A successful messaging paradigm must be as relevant between coroutines or POSIX threads as it is between lightweight kernel processes in a micro-kernel, or between separate activations in different domains of a vertically structured operating system (such as Nemesis [Roscoe 95], for example).

2.1 Open Distributed Processing Environment

The Reference Model of Open Distributed Processing (ODP) [ISO 95] standardizes a vocabulary—complete with concepts, rules, design principles and guidelines—that enables distributed systems to be described at different levels of abstraction without recourse to ad hoc terms such as "process", "task", "light-weight thread", "coroutine", or "activation". It provides one vocabulary to describe the computational objects that interact as well as a separate vocabulary to describe the parts of an operating system engineered to support them. These vocabularies are part of the computational and engineering viewpoints that, together with the enterprise, information and technology viewpoints, comprise the whole model.

For the purposes of this paper at least, the ODP Environment is defined to contrast with the Embedded Environment in a number of ways:

Open Distributed Processing Environment	Embedded Environment
Open development Application design can be shared among many developers	Closed development Application design is (normally) contained within one developer
Open applications There may be conformance points within applications that enable internal interfaces to be re-used by third parties	Closed applications Internal interfaces are not (normally) intended for third party use
Focus on non-local interactions Significant design and the implementation of related mechanisms exists to accommodate interactions that span different computers	Focus on local interactions Design and implementation tend to address interactions inside the same computer, non-local interactions may be ignored altogether
Long-lived interaction infrastructure Interaction infrastructures (e.g. CORBA) are designed to exist independently of the lifetime of individual distributed applications using it	Per-application interaction infrastructures Interaction infrastructures are not (normally) required to outlive the particular embedded application for which they are used
Expectation of complexity Application design is normally done in the expectation that the	Expectation of simplicity Application design is normally done in the expectation that

Open Distributed Processing Environment	Embedded Environment
application will be required to scale and will accommodate an evolution in the original requirements.	requirements and scale will not change following application deployment.

Table 1

Since many of these differences between the ODP Environment and Embedded Environment appear to be differences in requirement handling and design, and many of the choices made in the ODP Environment appear to conform with "best practice" it is tempting to deduce simply that the Embedded Environment ought not to exist distinctly and that embedded applications ought to be designed and implemented as they would be if it were in an environment such as ODP. An alternative, and perhaps more positive, point of view, which is taken in this paper, would be to accept the failings that omitting best practice might bring and consider whether there are benefits in exploring the design freedom implicit in the Embedded Environment.

The ODP reference model provides a basis for standardization in Open Distributed Processing and contains much more valuable material than just the definition of the computational and engineering viewpoints. Very much of it (for example, the computational model) has value when considered in non-open, non-distributed[2] environments such as the Embedded Environment, which is why this paper refers to it.

2.2 System Configuration

One of the attributes presumed of embedded systems is the varied but static nature of its components. In contrast to other types of operating system, components (such as processes and devices) are normally embedded statically in the system build rather than loaded dynamically on demand[3]. Because of pressure on resources redundant software and drivers are not normally part of a build, nor is the additional code needed to load or select configuration data. In consequence, whereas other operating systems may be instantiated in only one or a small number of basic configurations (which then extend and reconfigure themselves dynamically), the configuration of an embedded system needs to be externally specified for almost every use. In support of this requirement EEK is closely associated with a system-building tool ("Project"), which parses a system composition language. The language is used to describe the objects and interfaces that are available and their initial instantiation. Project uses the language to build a script that can then be used to construct the required system out of pre-built binary or source-code component templates.

2.3 A Model for Computational Object Interaction

One way to gain confidence that an interaction infrastructure is general purpose is to use a model that is accepted to be generally applicable (such as the ODP reference model) and show how relevant concepts in the model can be implemented using the infrastructure.

The ODP computational viewpoint isolates two distinct forms of interaction between computational objects, which might both, therefore, be expected in a "complete" infrastructure. They are: those provided at operational interfaces and those provided at stream interfaces. Both operations and streams are complex interactions composed of a series of simple atomic interactions called *signals* (operations involve the synchronous two-way remote procedure call-style exchange of signals whereas streams may involve a number of isochronous one-way exchanges of signals)[4].

Signals each conform to a specific signature that indicates its "meaning" and the type of data it carries. A structured *interface signature* that incorporates the signatures of each signal involved describes each interface. In effect, it also describes the protocol used: insofar as it dictates the sequencing and direction of individual types of signal. In the case of an operation the protocol is fixed: normally an *invocation* signal is sent in one direction and a *termination* signal is then sent in the reverse direction.

[2] Or not-necessarily-open, not-necessarily-distributed to be more precise.
[3] This is not to say that embedded systems are defined not to be able to extend themselves dynamically. They might be able to, but typically choose to do so only infrequently.
[4] This is not a tutorial on the ODP reference model, the actual content of the standard is being paraphrased and refined for the purposes of this paper.

ISO has adopted the Object Management Group's Common ORB (Object Request Broker) Architecture (CORBA) Interface Description Language (IDL) [OMG 99] as the standard language to use to describe operational interfaces.

In the context of a given interface signature a computational object must take one of the *roles* that have been defined for that type of interaction. In the case of an operational interface the role will be either "client" or "server". In the case of a stream an arbitrary number of roles might be defined. In principle, the interaction behaviour (and thus the supporting code) needed to implement an interface is determined exactly by the combination of interface signature and role.

In order to use an interface the ODP computational viewpoint defines the operation of binding, which attaches complementary interfaces together. For example, given an operational interface signature it might attach a client version of the interface to a server version of it. An interface supported by some other computational object can be used once a local *binding* to it has been created. A traditional way to support such a binding operation is to use interface references [van der Linden 93, ISO 95] that locate the participants needed. When interface references are mobile they can be passed from object to object and may be able to be stored in long-term data storage objects.

2.3.1 Signal Bindings

In EEK signals are implemented as the basic form of interaction and they are used to support both operational and stream interfaces. A *signal binding* supports an interface for the delivery (and receipt) of a one-way signal. More complex operational or stream bindings can be made from a number of these more primitive signal bindings.

In designing a signal binding the invocation mechanism (the means whereby causing a signal results in a signal notification) must be chosen, and in particular:

(1) how a signal can be caused (by a *causer*); and,

(2) how a signal can be notified (to a *notifiee*).

In accordance with its design goals (and Occam's Razor) the simplest possible design was chosen for EEK: in which causing a signal is implemented by calling a function and notification of invocation is implemented by being invoked as a function. This means that the most trivial implementation of a signal binding is null: the cause and the notification of a signal can be accomplished in the same action (a function call).

One of the advantages of such a design is that signal (and thus operational and stream) interfaces are potentially very cheap to implement and can therefore be used freely throughout an operating system. One of the main disadvantages of such a simple implementation of a binding is that it is often inappropriate. Consider, for example, the implementation of an operation supported by a "server" function: this function will be invoked to notify the server of an invocation signal and, after some processing it will need to cause a termination signal. Using a simple implementation of a signal binding this would result in calling a function in the "client", which will perhaps need to invoke the server again and thus continue a recursive regress that is likely eventually to cause a stack error. However, whilst a direct call is the simplest binding mechanism, and occasionally is of great use, it is certainly not the only signal handling policy as we will discuss below.

2.3.2 Interface Signatures

In the ODP reference model both operation and stream interface signatures are intended to be structured descriptions of the signals that require notification and that can be caused at an interface. There may be many uses for such a description including "type checking" whereby the description of a provided interface is compared with a description of one that is required – to check whether it can be used. Such run-time type checking is of value in a kernel to increase the robustness of modularly constructed systems.

Note that, because there are different roles that an interface signature caters for (e.g. client or server in the case of an operational interface) in general, the interface provided by those with one role in a binding will be different from that being provided by others. Thus, for example, given a particular operational interface, the interface that the client requires of the server is not the same as the interface the server requires of the client. The difference is in whether the required interface is to cause, or be notified of, the individual signals from which the interface is composed. Furthermore it should not be assumed that an object with a particular role would have a monopoly on taking the initiative in seeking others in a binding. It is possible, for example, for a server to seek clients actively – in which case the interface signature it requires will be complementary to the one that a client would need of a server.

"Type checking" involves ascertaining, at minimum, that the provided interface accepts all the signals (of the appropriate type) that the required interface would and that it does not generate any signals that the required interface would not. A more elaborate check might also take into account the semantics associated with the interface or perhaps the quality of service associated with signal delivery etc. By and large, however, the run-time implementation of the check will involve verifying a relation that should hold between two interface signatures. In practice, only two operations on interface signatures will take place:

```
BOOL if_subtype_of(INTERFACE_TYPE required, INTERFACE_TYPE provided)

INTERFACE_TYPE if_combine(INTERFACE_TYPE interface1, INTERFACE_TYPE interface2)
```

These functions are supported in EEK. Unlike a large distributed system it *is* practically feasible to change the implementations of these operations (e.g. altering the way an `INTERFACE_TYPE` interface signature is implemented) to suit each embedded application. In EEK the standard implementation is extremely simple and, again, makes use of the static nature of an embedded system. In systems which do not generate new interface definitions dynamically the full implementation of an interface signature, that would be available from a structured description incorporating type information about each signal, is not necessary at run time. In essence, the `INTERFACE_TYPE` × `INTERFACE_TYPE` relation can be pre-calculated and embedded into each computational object. The disadvantage of such an approach is that many implementations of the relation will not allow the introduction of new kinds of interface whilst the embedded application is running (although some may). The advantages include:

- efficient implementations of the two operations above are possible (e.g. with a complexity independent of the number of signals involved in an interface); and,
- more complex issues, such as the semantic nature of interfaces[5] and environmental (e.g. Quality of Service) guarantees can be taken into account during the construction of the relation without imposing any additional run-time overhead.

Because it is assumed that it would be practicable to implement an alternative representation of this relation throughout an embedded system prior to its re-instantiation[6] and thus improve its features when necessary, it is possible to default to an implementation that concentrates on the efficiency rather than general applicability. Thus the default in EEK is to maintain an offline register of primitive interface components and to represent each `INTERFACE_TYPE` as a set of these – the set being implemented as a simple fixed sized bitmap. This allows the two `INTERFACE_TYPE` operations to be implemented using AND and OR operations respectively.

This approach would become unwieldy should the number of primitive interfaces become very large because the size of the bitmap is dependent on their number. In addition, it will not allow the retrograde run-time introduction of interface types "more primitive" than those already selected into which any existing interface types ought to be recomposed. In deference to what was said above, however, it does allow a limited possibility to extend the number of interfaces extant in a system at run-time insofar as new combinations of primitives can be used, and new primitive interfaces might be allocated positions in the bitmap if it is initially deliberately "over-sized".

Although not its primary motivation, one of EEK's design guidelines—to re-use existing interface types wherever possible—happens to be well rewarded by this approach.

2.3.3 Operational Interfaces

By far the majority of interfaces in EEK are operational (as opposed to stream) interfaces. In terms of the ODP computational model operations may be either an interrogation (in which a termination follows as a

[5] For example, although an interface signature for copying and deleting things may be structurally identical to one for renaming and deleting things their semantics are not compatible and it may be a mistake to allow the latter to be used where the former is required.
[6] In the case of a single node system this corresponds to recompiling, re-linking and re-deploying.

consequence of an invocation) or an announcement (in which no termination results). The latter requires the definition of a single message to be used to carry the invocation signal and the former may require an arbitrary number of additional signals to convey the different terminations. All of the messages involved in the same interface are described to Project in an interface definition file. Project then provides a unique value associated with each signal type and also generates a data structure that represents the fields of the message. Code that marshals and unmarshals messages uses the signal type value and data structure provided. Currently Project supports only "C" or "C++" data structures.

Each interface definition file specifies the type of the interface for each role in which it is expected to be required. It does this either by composing it from a combination of other interface types (the message definitions are not required in this case) or by indicating one or more primitive interface types to which it conforms.

In interrogations, because EEK supports call by reference there is a substantial saving in complexity (particularly in memory management) to be gained by using the message data area used for the invocation to return any of the terminations. In order to assure a consistent message size that can be used for both purposes it is normal to use the same message definition for both the invocation and termination. Project supports this mode of use by allocating both an invocation signal type value and a termination signal type value for each message definition. The termination value will also be allocated even when the message is intended to be part of an announcement.

The way in which memory for messages is to be managed is an important detail, which can be recorded in the interface definition file, but there is no run-time support to enforce or prevent any particular combination of memory management operations on messages. The dominant protocol in interrogations is to return a termination in the same message area and thus leave memory management entirely in the hands of the sender of the invocation.

Binding an operational interface requires:

1. the object in a client role to obtain a signal binding for the delivery of an invocation signal to the server; and,
2. (in the case of an interrogation) the server subsequently to obtain a signal binding for the client (who invoked it) for the delivery of a termination signal.

There are a number of ways for a client to obtain an initial signal binding to the server, which we describe below. Although these methods are also applicable to the server, a server will normally choose to use the feature of EEK's signals that enables it to derive a return signal binding from an incoming message. Each message has an optional field that can hold a return binding specified by the signal sender (see Figure 1). This binding may need transforming as it traverses a system and this is discussed below.

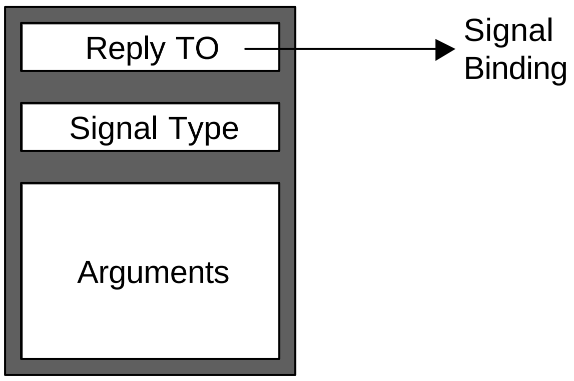

Figure 1: Major fields in a message

2.3.4 Stream Interfaces

As indicated above, the interface types required by one of the participants in a binding will vary according to the roles the others in the binding are to take. To this end a typical stream interface will require a number of interface types – up to one per role. Because of the variable nature of the protocol in a stream interface these types need no longer be simple inverses of each other, as they are in the case of an operational interface: for example, one participant may cause signals that are notified to another but be notified of signals that are caused by a third.

Other than making it unlikely that the additional signal types defined by Project for use as a termination will be used, stream interfaces do not impose any additional requirements on Project.

Binding a stream interface requires each participating computational object to obtain signal bindings to the required subset of other participants consistent with interface types appropriate to their roles. Each such signal binding can be obtained only in the ways that would also be applicable to an operational interface, and in this respect EEK needs no additional support to enable the use of streams. One problem, however, is exacerbated in the case where many individual signal bindings have to be made at a number of different points before the whole stream binding can be said to exist: and that is the overall co-ordination of creating and closing down the individual bindings and of monitoring errors in any isochronous flows involved. A single computational object, the control binding, is expected to handle these tasks and provide a unified interface for the stream.

An unacknowledged signal type, `WRITE_FREE`, is provided for use between the source and sink of an individual flow within the stream. Such messages are allocated at the sink and freed either en route to or at the sink. A local `SINK_LINK` signal type requests a flow source to open a flow and provides a storage object from which signals should be retrieved, with a given quality of service, to a sink and a `SINK_UNLINK` signal type requests its closure. EEK provides a library supporting the use of these signals and another for creating a simple stream with just one flow.

3 Signal Transport

One of EEK's design goals is to provide intercommunication primitives that would be appropriate for the generic case. Therefore, even though generality will not necessarily be a feature of its implementation, some of the aspects of that general case need to be established. The principle cases to consider are those in which natural boundaries between the "causer" and the "notifiee" of the signal make the simplistic function-call style binding inapplicable. These boundaries include:

invocation mechanism boundaries

- in which there is no existing type of invocation mechanism that common to the causer and the notifiee (e.g. where there are invocation mechanisms that allow "threads" to talk to each other and mechanisms to allow "library objects" to interact but none that allow threads to talk to library objects); and

invocation medium boundaries

- in which no existing message reference namespace is commonly available to both causer and notifiee (e.g. causer and notifiee use different non-overlapping memory domains).

An invocation interceptor is an object that supports the transport of signals across either an invocation mechanism boundary or an invocation medium boundary. In general there may be a string of an arbitrary number of interceptors that need to be used between a causer and the corresponding notifiee. There may be many ways to represent such a route and many different functions capable of transferring a message along that route. In EEK the signal binding is represented as a triple B=<**signal**, **route**, **abort**> where **signal** is the function that is used to cause a signal, taking a message and a route as arguments; **route** is the route appropriate to the function and **abort** is an EEK event (described later in section 5) which is used to signal the failed availability of an onward route (see Figure 2). The most important characteristic of an event here is that it provides a signal to all those who express interest in it, whenever the event signal is caused.

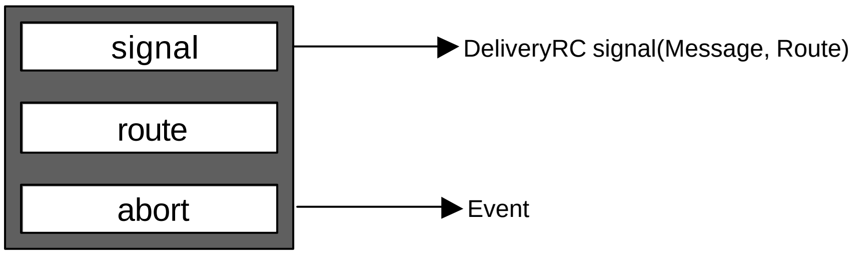

Figure 2: Major components of a signal binding

3.1 Forward Route

In general an invocation interceptor is an object that responds to a **signal** function by finding and using the next signal binding B_2=<**signal$_2$**, **route$_2$**, **abort$_2$**> that it can derive from the context in which it was notified of the incoming signal and the **route** supplied. The detail as to how this is done will vary depending on the type and exact function of the interceptor. Since the derivation of B_2 may sometimes require significant activity on the part of an interceptor, it may choose to cache the <route, context> ? <signal, route, abort> mapping it makes.

3.2 Failure of The Forward Route

An interceptor will also respond to the prevailing conditions of its onward routes (either continuously or on each transmission attempt), indicating a delivery failure for transmissions that require the use of an invalidated route. If the onward route was itself obtained as a signal binding, its associated abort event can be used to provide asynchronous indication of the route's failure. Similarly if the interceptor has provided a binding that relies on one of its routes, the abort event provided in them can be caused when the route is found to be invalid. Indeed if every interceptor in a route obtained a signal binding for each binding's onward route it is possible for the abort event at the destination to be visible to the holder of the analogous binding at the source (having been reflected through each intervening interceptor) even though the abort may occur asynchronously with respect to the transfer of signals along the route. Where this asynchronism is desirable it can clearly be implemented as above, which requires:

- that the signal binding, initially local to the notifiee, was delivered to the causer via each of the interceptors on the route indicated in the binding local to the causer; and,
- that each interceptor holds a different forward binding (with its own abort event) for each binding for an incoming signal.

There is one case in which asynchronous delivery of an abort event is required. This is the case when there are no interceptors at all and a direct "functional" binding is in place. If the binding becomes invalid (for example the route ceases to be a valid address) this cannot be ascertained by attempting to use the binding to cause a signal and observing whether a delivery error occurs. The indication not to use the invalidated binding can, however, safely be delivered asynchronously via the abort event.

3.3 Return Route

Because messages contain a return binding, interceptors must transform them as they pass through, for example by allocating storage to represent the previous binding B_{rev}=<**signal$_{rev}$**, **route$_{rev}$**, **abort$_{rev}$**> with handle **route$_{rev2}$** and replacing it with B_{rev2}=<**signal$_{rev2}$**, **route$_{rev2}$**, **abort$_{rev2}$**>, where **signal$_{rev2}$** is a function that uses this handle to find the stored previous signal binding. Setting up a reverse route across a system back to the causer can be expected to use resources in the system, and some mechanism to recover these resources is necessary.

A simple mechanism for resource recovery would be to delete the reverse route information after a fixed time (which yields a "soft state" style of connection). A more sophisticated mechanism might rely on an explicit abort event generated at or nearer the notifiee. Additional mechanism might use signals generated by the abort events of stored reverse bindings nearer the causer. If the route is not liable to be re-used, there may be some benefit in signalling an abort event in a message's return binding once the message is no longer needed.

3.4 Object Creation and Deletion

One of guidelines associated with the use of EEK is the re-use interface types where possible. It is sometimes possible to decompose individual operations in a way analogous to currying a function: an operation is provided with a subset of the required operation's arguments that creates (and returns) a new object supporting an interface containing an operation with the remaining arguments. For example an interface to a file system might originally be envisaged with two operations: one `read(FileName, Buffer, BufferSize)` and another `write(FileName, Buffer, BufferSize)`. However this can be replaced by two interfaces: one that supports the single operation `opennamed(Name)` that returns an interface reference; and, another (to which the new interface conforms) providing `read(Buffer, BufferSize)` and `write(Buffer, BufferSize)`. Of the two new interfaces the latter is very generally applicable to many things that can be read and written, and the former is very generally applicable to many things that can provide interfaces bound to a name. In order to support such decomposition (and for other reasons) EEK must be able to return a signal binding as the result of an operation.

As described above, signal bindings need to be transformed as they travel (through interceptors) from one invocation domain to another. Every interceptor in the path of a message some of whose arguments may be signal bindings would have to have sufficient knowledge of each message's semantics to determine the position of each binding so that each could be replaced by appropriate transformations, and this is an implementation expense that could otherwise be avoided. Accordingly the use of a signal binding as an argument in a message is deprecated in EEK. In its place two distinguished interface types OPEN and CLOSE are identified containing the operations `open()` and `close()` respectively. Typically these interfaces will be used as part of the definition of a new interface with operations based on the definitions of `open` and `close`. Normally one interface will include an `open` operation that returns an interface binding which includes a `close` operation. The definitions parsed by Project for message formats allow one message type explicitly to be based on another, so Project can generate message numbers that make their parentage obvious and interceptors handling such messages are able to identify them as being based on `open` or `close` operations.

In the message carrying the termination signal of an operation the "reply to" field normally is of little consequence. In the case of an `open` termination, however, it contains a new binding to be delivered to the caller. In the case of a `close` termination it contains an empty binding. Because these messages can be identified in interceptors, the relevant transformation of the new binding in an `open` can take place (including the allocation of relevant resources). Such resources can be removed on identification of a `close`. The transformation of the termination "reply to" field in this case is identical to the transformation of this field required in an invocation in order to create a reverse route (discussed above). In fact, interceptors that perform the relevant mapping on every "reply to" field in every message they see (no matter whether or not it is an `open` and whether they are an invocation or termination) will function correctly.

3.5 Abstract Syntax Boundaries

In addition to boundaries placed by invocation mechanisms and the invocation medium another boundary is formed when the representation of types (transfer syntax) in a signal (i.e. the argument types of a message) can vary.

An interceptor that bridges this kind of boundary must be able to translate between the syntax used by the invocation mechanism on the causer's side to the syntax used by the invocation mechanism on the notifiee side (for example, the local syntax of the causer's computer to the transfer syntax encoding used on a network). One of the most important abstract syntax boundaries is one between a local (native to a network node) syntax and a transfer syntax (used when transmitting a signal across a network).

It is reasonable to expect such interceptors to know the rules with which different types are encoded in the two syntaxes with which it is dealing. Interceptors must, however, also know which types any particular message uses and into what kind of data structures they are assembled. The same description of a signal type could be used by all the transfer syntax interceptors on a signal's path if a suitably generic abstraction of each of the transfer syntaxes were used. This kind of description of a message is called its abstract syntax. Ideally interceptors require a means to obtain the abstract syntax of any message, in order to perform their function.

The minimum that is required in a message to identify its type is an identification code[7] (referred to as "signal type" above). This code can then be used to index a table of abstract syntax definitions of all the messages in the system. Some means of generating such definitions is required, however. Currently the information held in an interface definition file processed by Project includes a definition of an identification code and the abstract syntax for every message. However, the notation used to describe the abstract syntax is currently "C", which has many disadvantages when used as an Abstract Syntax Notation (ASN). In addition Project does not encode this information into files incorporated into system builds, as would be required to build something analogous to CORBA's interface repository.

These are areas for further work in EEK that would be required if abstract syntax boundaries are to be brooked. The use of a "proper" ASN and a mapping onto C could involve significant development, but would involve changes to existing code only for (remote) interactions whose existing interfaces could not be re-expressed in a new ASN. Initially, however, it is sufficient to show only that the chosen intercommunication primitives would be appropriate for the generic case, in this event by proposing that an abstract syntax interceptor could be built.

4 Trading

The previous section explained how a signal binding should be involved in interacting with remote interfaces. It also described one method of transferring signal bindings from location to location, by returning them in messages based upon the `open` operation. In principle this means that, given the right functionality, any binding that might be required could be retrieved once there is some binding available that responds to an `open` correctly. However, there is still the question as to where the initial binding comes from.

The ODP reference model defines the notion of "trading" to accomplish this aim and this function has been standardized by ISO and the International Telecommunication Union's Telecommunications section (ITU-T) [ISO 96] which is a successor of the work in ANSA [APM 93]. The most important elements of this function are: the ability to export an interface reference to the trading system; and then, to be able to import it again (possibly from a different invocation domain) later.

In EEK trading is accomplished by programming objects ("traders") characterized by methods to:

- import a local binding (i..e. one usable in the current invocation domain) given an INTERFACE_TYPE and a name given that it is "closer" than a given distance;
- create, modify or delete an entry in a trader where an entry is either:
    - ♦ an interface: with a trader-local signal binding, its INTERFACE_TYPE and a trader-local name;
    - ♦ a link to another trader: with a trader-local name, a reference to another trader and its "distance" away; and,
    - ♦ an alias to another entry (which may be in another trader): with a trader-local name prefix, a reference to another trader and a replacement prefix to use in the other trader.

Many traders may be available and each could be implemented differently, however EEK does provide both a very simple "standard" general-purpose implementation and a distinguished "node-wide" (available throughout the node) trader, which is always present. It also provides a trivial library that provides import, export, link, and alias functions in addition to a few others dealing with updating and removal.

Links and aliases enable a web of traders to be integrated to accomplish the trading function. When seeking an interface with a given name the mesh is traversed according to prefixes in the name until an interface reference with the required type is found, which is then returned, or failure occurs.

There are a number of apparent differences between this implementation of a trading function and a traditional implementation. The following will be discussed in particular:

1. the trading function is not necessarily provided by trader computational objects (with signal-based interfaces of their own); and,

[7] Similar to the "type code" used in the CORBA protocols.

2. the trading function provides for the export and import of signal bindings and not interface references[8].

4.1 Trader Computational Objects

To help satisfy the "cheapest possible implementation in an embedded system" goal a library-based implementation of the trading function was chosen. It provides for ease of implementation, small size and efficiency at runtime. In most trading scenarios in the literature the trading function involves traders that are themselves computational objects able to interact across networks in a loose co-operative federation. In order to show the other part of EEK's goal (that intercommunication primitives are provided that would be appropriate for the generic case) it is important to demonstrate that the proposed interaction scheme, using signal bindings, can scale to similar systems (in which traders accessible via their own, possibly remote, signal interface can be used).

Clearly it is possible to create a trader computational object with an appropriate operational interface. It is also possible to create a trader implementation that uses such a computational object to perform the operations required for each method. It is not immediately apparent however, that it is possible to retrieve relevant or efficient signal bindings from such objects. Signal bindings are relevant and usable only within their home invocation domain – a binding relevant to one domain, stored in the trader object's domain and made available in a third domain provides a variety of opportunities for corruption or inefficiency. For this reason, in EEK the signal bindings that traders hold are always local to their own invocation domain.

The `import` operation of a trader computational object should be based on `open` – returning the imported binding in the same way than an `open` operation will. This will ensure that the binding available at the trader object (which, by definition, is the same location as the interface the binding gives access to) is correctly translated to a binding usable at the operation's invoker.

A given binding can only be exported to local traders and this might appear to introduce a restriction on the availability of interfaces. It is, however, possible to effect an "export" by placing aliases, or links, in more remote traders to a local trader. The importing user of the trading function would be unable to distinguish whether or not traders are restricted by this implementation detail.

4.2 Interface References

The combination of a trader name and the name for an object in it, being the information required to insert an alias in another trader, can be treated as an data type in its own right and can be used as an "interface reference". Note that the name for a trader must be locally relevant, so that if it is moved from one place to another within the same name domain the name need not be transformed, but if it is moved outside that domain it will require renaming in order to refer to the same trader object. Subject to the need for this renaming, such interface references can be stored within a system and can be "exported" to other traders (insofar as creating an alias in them would have the same effect).

In general an EEK interface reference is a hierarchical name, led by a local trader, in which each successive part of the name identifies a computational object within the context of the trader so far identified. Thus if an external name E is found in external trader L/T, where L is the local trader and T is the local name for the trader, the interface reference L/T/E can always be used for it locally. However, if L/T is known to be part of the same name domain as L then L/T/E identifies the same object as L/E and the interface reference L/E can be used instead.

Because a signal binding provided by an `open` includes a route from the object to which it refers (the one via which the `open` termination was returned), there are benefits in using the information in intermediate trader computational objects directly, rather than allowing traders to request an import recursively. Consider the situation in which a local trader has an alias to an intermediate trader object that itself has an alias to a final trader object: if the local trader requests an "import" from the intermediate trader object the binding will be returned via the intermediate trader – and will therefore have the intermediate location in the binding's route. This can result in the "triangular routing" problem: whereby signals to the located object, which is potentially

[8] That is, the thing imported using the function is not subject to a separate binding operation before the interface it refers to can be used.

nearby, always travel via the intermediate trader which may be some distance away. If the local trader retrieves the trader name for the intermediate alias directly (including any name transformation needed) and then uses that to retrieve the binding from the final trader the returned binding need not have the intermediate trader as part of its route when all the traders are in the same name domain.

5 Events

Event counters are described in the context of their use in the Fawn operating system described in [Black 95] and were used in the Nemesis operating system developed as part of the ESPRIT funded "Pegasus" project [Roscoe 95]. [Black 95] defines the operations that event counters support, and provides a number of examples in which many extant styles of concurrency control are implemented using them[9]. An event counter "e" is defined to support operations with the following semantics:

read(e)

- This operation returns a value of the monotonically increasing counter associated with the event, as it was at some time between the time this “read” operation was invoked and when it terminated.

await(e,v)

- This operation blocks the caller until the event counter reaches or exceeds the value “v”.

advance(e,n)

- This operation increments the value of the event counter by “n”, which may cause “await” callers to execute or become runable.

Two types of interface to events are implemented: an operational interface, which supports operations `event_read`, `event_await` and `event_advance`; and, a procedural interface. The procedural interface does not use signals and has a small performance advantage.

The operational interface is supported by a library function that returns a local signal binding to access a given event (that is created in C programs as a data structure with type `EVENT`, which essentially keeps the current event counter value and the set of pending `await` invocation messages). This binding may then be used to access any of the above operations.

Note that the value awaited need not be the “next event”, and that the value of the event counter may exceed the value awaited by the time an await operation completes. These properties allow `EVENT`s to be used for relatively loosely coupled synchronization in which event generators and event consumers can each operate without necessarily requiring the other to be scheduled between events. When the processor is busy this can save unnecessary context switches.

6 Signal Ports

As implied above in section 2.3.1, signal reception at a signal binding is indicated by the invocation of a function of the receiver’s choosing. This may be appropriate when dealing with some signals (for example, those that can be serviced instantly), or it may provide a convenient opportunity to demultiplex signals into an application’s data structures (e.g. message queues for different functions). However, when the signal is required to be handled synchronously with its notifiee, the signal must be stored in some intermediate location until that synchronous opportunity arises. A *signal port* is a general-purpose representation for this state (illustrated in Figure 3). It includes the signal binding through which signals are delivered and additional functions to poll, read and time-out any pending signals. A small library of functions allows signal ports:

- to be initialised with a signal binding;
- to define the functions used for polling, receiving and setting time outs for a particular variety of signal port;

[9] It also explains their implementation as part of the fundamental scheduling infrastructure in Fawn. The implementation here merely duplicates event counter operations, it does not form part of EEK’s task scheduler.

- poll:
  to inspect whether or not a signal is available for reading on the signal port;
- read:
  to (suspend execution, wait then) return the first signal available on the signal port, or an error code; and,
- timeout:
  set a timeout for the next read.

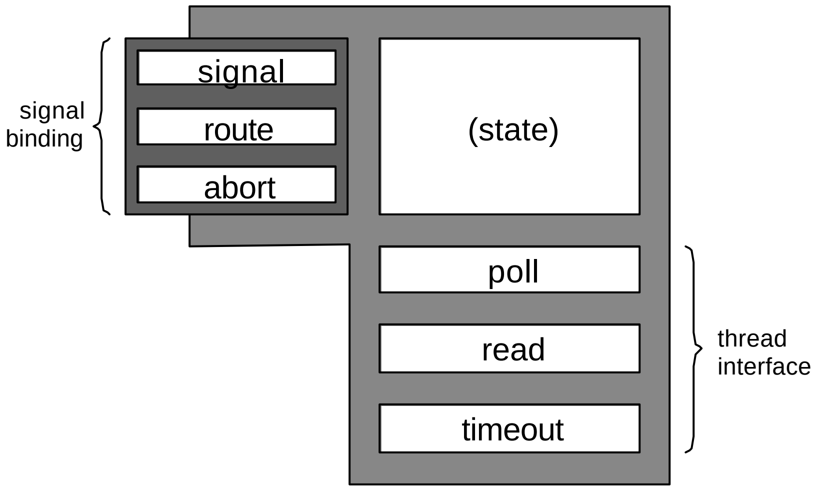

Figure 3: Major components of a signal port

Different signal ports can be defined for operation not only in a wide range of concurrent execution environments but also to support a wide range of signal handling policies. From the perspective of a particular thread of execution signal ports have two common roles: client synchronization – to synchronize with the notification of an operation's termination signal in an operation exchange; and, service synchronization – to synchronize with the notification of an operation's invocation signal once processing due to a previous invocation is complete.

6.1 Client Synchronization

Synchronous exchanges, such as those that might be required during the invocation of an operation on a computational object, can be achieved through the use of a signal port uniquely allocated for the purpose. The sequence of behaviour is:

- the client thread obtains a unique signal port, initialised for synchronous working;
- the client sends an invocation signal to the server quoting the unique signal port for any reply; then,
- the client waits for a termination on the signal port.

Since a thread can wait on only one signal port at any one time a signal port used for this purpose can be pre-allocated, which will removes a certain amount of overhead that might otherwise be associated with this algorithm.

6.2 Service Synchronization

Self-synchronizing exchanges, such as those that might be required during the sequential servicing of operation invocations on a computational object, can be achieved through the use of a single "work" signal port that is capable of retaining and ordering a set of invocation signals. Typically the repetitive sequence of behaviour would be:

- the server waits for an invocation signal on the "work" signal port;
- the server performs the required processing in order to define the termination signal; then,

- the server replies with the termination signal to the “reply to” field in the message representing the signal.

In a typical server thread the signal binding associated with the signal port would be the one through which it advertises itself (e.g. in a local trader).

6.3 Signal Handling Policies

A signal port describes not only how a process stored signals destined for its thread, but also describes the way in which signals are placed there. Signal bindings can incorporate an arbitrary amount of state identified by the route associated with their signal binding. This can be used by the notification function to accomplish many different goals. Although the architecture of the port thus enables a variety of signal handling policies to be met, the signal interface the port provides remains standard, as does the interface the port presents to a thread using it.

6.3.1 A Mailbox

A signal “mailbox” is implemented trivially by using the port’s signal binding’s route to identify a variable capable of holding a single signal. The signal binding’s signal function returns a delivery error if the variable is in use or otherwise places the signal in the variable[10]. The port’s waiting function returns the content of the variable when it is available.

6.3.2 A Queue

A signal queue is implemented by using the signal binding route to identify a queue and the signal function to append each message to the back of the queue. The port’s read function takes the first message on the queue.

6.3.3 Listening on Many Queues

A number of different signal bindings are distributed to different causers. Each binding has its own signal queue and all signal bindings share a single mailbox and a bit map containing one bit for each signal queue. When a message is transmitted the signal is appended to the relevant queue, the bit corresponding to it is added to the bit map and the bit map is placed in the mailbox[11].

The port’s poll function checks the bit map to see if it is non-zero. The read function uses this value to determine which queues have messages available, choosing randomly among them and adjusting the bit map if a queue is emptied.

Note that this type of signal port could be constructed by composing mailbox and queue signal ports.

6.3.4 Prioritorized Signals

A number of “real time” kernels provide a mechanism to send high and low priority messages. Higher priority messages arrive before lower priority ones. This can be implemented as a multi-queue signal port as described above with one queue per priority level. The signal function of the port’s signal binding finds the priority of a message and appends it to the queue for that priority. The read function reads a signal from the highest priority queue whenever a signal is available.

6.3.5 Load Sharing

A set of threads can be identified by a port’s signal binding’s route. The corresponding signal function can then choose either the “next” task in the list to which to send an invocation signal or, perhaps, the first one that is waiting on its queue. In this way a number of instances of the same process can be active to service the same stream of operations.

[10] If scheduling is involved the signal function might also indicate that the thread associated with the port has become runable.
[11] If scheduling is involved then making the bitmap non-zero might also be used to indicate that the thread associated with the port has become runable.

6.3.6 Functional Discrimination

As in the load sharing example above, a number of threads are included in the same signal binding route, but in this case, each is associated with a different group of signal types. The threads co-operate to service the full range of signal types but each undertakes only a subset of them. The signal function associated with the signal binding demultiplexes incoming messages to the appropriate queue based on their signal type.

Note that the user of this signal binding sees no distinction between this kind of binding and any other. Effectively the detail of the way in which the system is structured behind the signal binding to service his requests is transparent to him.

Naturally the same strategy can also be used to distribute signals to queues that are all accessed by the same thread.

6.3.7 Immediate Response

Some types of signal require a minimal amount of processing (for example they may be used to set an integer representing a debug level). The signal function associated with a signal binding potentially can isolate these messages and execute their associated function immediately rather than passing the message on to a queue for later processing.

This might also be appropriate for messages that need a minimal overhead in their handling for which the expense of a context switch is too great, or in which no other context (e.g. process) is available.

7 Some Uses of Signal Bindings in EEK

EEK incorporates a third party scheduler as one of the modules incorporated into every system build. The one chosen was µC/OS [Labrosse 92], which is publicly available, small, almost entirely written in C, and has been implemented on a wide range of "embedded" processors. It supports a priority-based pre-emptive scheduling regime in which a limited number of processes are each identified by a unique but variable priority and it provides primitives very similar to a number of more commercial offerings. Inter-process communication is supported with fixed length queues, mailboxes and semaphores and it imposes no restriction (or recommendation) on Inter-Process Communication (IPC) formats etc.

Although the scheduler is relatively compact and efficient this is achieved at the expense of a fixed process limit (63) and a requirement for unique priority values. However, since almost all of the functions provided by µC/OS are invoked through the signal binding and signal port abstractions it is a relatively simple matter to replace it with another scheduler should these restrictions become onerous.

7.1 Signal Bindings in Processes

By default every µC/OS process created is accompanied by two signal ports, each constructed around a µC/OS message queue. One (the mailbox) is intended for use in client synchronization and the other (the work queue) for use in server synchronization. On creation the signal binding for the work queue is placed, along with the process's name and its interface type in the node-wide trader.

A system description file parsed by Project determines the initial set of processes in a system. Project supports the specification and instantiation of arbitrary system objects – allowing different types of object initialisation to be defined. In fact, a µC/OS process is an object entirely defined in one of the "modules" parsed by Project, and processes in a system build could not be supported without it.

7.2 Signal Bindings in Other System Objects

Other modules support other system objects, for example generic "devices", or more specific objects such as UARTs.

The instantiation of system objects may or may not involve the creation of signal bindings and their export to a trader. In the case of devices, for example, each instantiation registers an instance of the predefined device API for a named type of device. In the case of a UART the instantiation of the API appropriate to the named kind of UART is found and an entry is placed in the device trader which is linked-to from the node-wide trader for names prefixed with “dev/”. This trader generates signal bindings that map on to the device APIs when imports are requested. Thus an import request from the node-wide trader using the name “dev/tty2”, with the appropriate interface type, might obtain an interface to one of the system UARTs. Note, that in an alternative system build, in which there was a process named “dev/tty2”, the same import might provide access to an identical interface which differs only in the nature of its implementation.

7.3 Standard I/O using Signals

The C library used in EEK makes use of traders to implement the standard input/output functions. Each instance of the C library (normally each instance is tied to a different process, but they may be tied to arbitrary system objects) contains a *local trader*. Initially this is set to the system’s node-wide trader, but the library user can subsequently change it. Streams are implemented very simply by using a `FILE` to hold references to a small number of functions that are used to read a block of data, write a block of data, return the stream’s size, and close it etc. A small number of stream implementations exist, one of which makes use of “block” interfaces and another of which makes use of “pipe” interfaces. In either case the associated `INTERFACE_TYPE` is constructed from two more basic `INTERFACE_TYPE`s: one for reading and one for writing. Appropriate block and pipe interface types are constructed depending on the requirements to read or write. A signal binding is sought using a name which is imported from the local trader under first one interface type and, if that fails, the other. A new stream is opened using a stream implementation appropriate to “block” or “pipe” interfaces depending on which interface type succeeded. The close associated with the stream will invoke the `close` operation on the signal binding if that interface is supported.

7.4 An “Open” Trader

The scheme above allows a fixed set of streams to be implemented easily – each file being named and exported into the local trader – but it does not conveniently provide the kind of interface traditionally associated with a file system – since it would appear to require every “file” to have its own entry in the local trader; nor does it provide an interface in which parameterised stream names can be used – since stream names for each possible option would need to be available in the trader redundantly. These are the problems of hierarchy and dynamic instantiation respectively. The hierarchical problem could be addressed by creating a hierarchy of linked traders to service the whole set of files, or it could be addressed by a dynamic search in some other data structure, for which a solution for the dynamic instantiation problem would be required.

In order to accommodate the dynamic instantiation of signal bindings to service “stream names” there is an *open trader* in EEK. This import-only trader is instantiated with a specific signal interface that supports the `objopen` operation that is based on the `open` operation extended to include a single name as its argument. When any name is requested for import from this trader the `objopen` interface is used to retrieve a new signal binding for that name and, if successful, this is returned as the result of the import. An in-store filing system module, for example, implements an `objopen` interface that returns a “block” interface to named files held in memory and creates an open trader from it. It simply links this trader into the node-wide trader after some appropriate prefix to implement the filing system.

7.5 Bridge Interceptor

Above the (theoretical) operation of an interceptor was described. They allow signal bindings appropriate in one invocation domain to be mapped into signal bindings for another. Because EEK is a single memory domain kernel there are no internal invocation medium boundaries, but there are invocation mechanism boundaries. The most useful interceptor is one that deals with the export of the simplistic intra-task procedure based signal binding so that it can be used as a inter-task message passing binding. The interceptor that supports this function (and more generally, the export of any internal interface through an existing signal port) is called a *bridge interceptor.*

The description of an interceptor in section 3 implies that reply signal bindings in messages might be translated dynamically as they are forwarded. However, because (in EEK) the “export” invocation mechanism is valid wherever the local one is, a significant efficiency can be gained by using export signal bindings for message replies all the time and avoiding the dynamic translation.

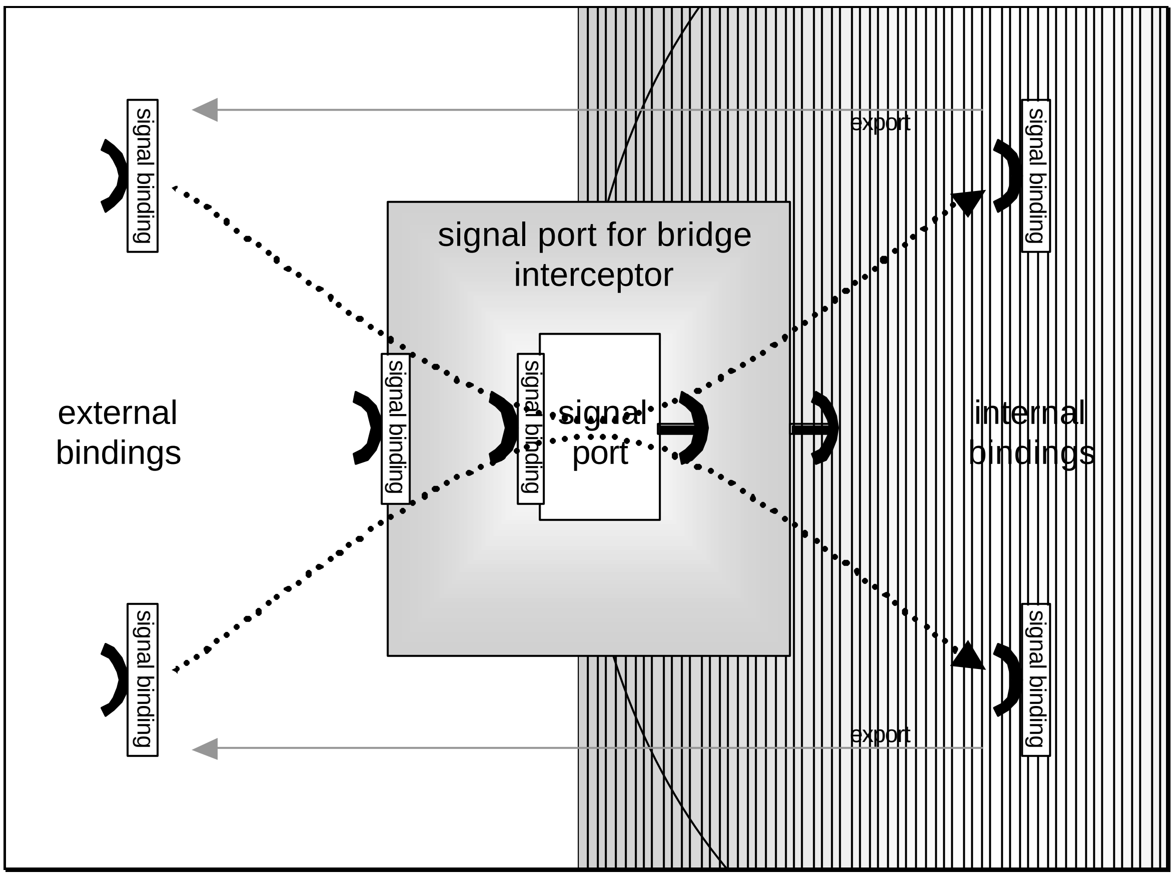

Figure 4: bridge interceptor

Because it is convenient for a server thread to receive all of its operation invocations on one signal port, a process normally supports only one signal binding. A bridge interceptor provides a signal port instantiated from an existing signal port that allows any number of external signal bindings to be generated from signal bindings internal to the thread's invocation domain. For example, if an internal signal binding might involve a direct call to a function local to the thread, an "export" function supported by a bridge interceptor will create an external signal binding that can be used in the wider invocation domain incorporating other threads. Bridge interceptors conveniently allow threads to support interfaces based on `open` (in which a thread may have to support an arbitrary number of newly created `open`ed interfaces).

In normal operation a thread will replace an original signal port by a bridge interceptor into which the original is bound. The external signal bindings it creates includes a

- **route**: a reference to the signal binding of the original signal port and the internal signal binding on to which it should map; and,
- **signal**: a function that associates the internal signal binding with the message before invoking the original signal binding.

The read function of the bridge interceptor calls the read function of the original signal port until a message not associated with any original signal binding is found, which it returns. The other messages are forwarded to the associated (internal) signal binding.

The standard bridge interceptor in EEK implements the association of messages with a destination internal signal binding using a special "destination" field in a message (which would not otherwise be necessary). This has the advantage of making bridge interceptors extremely simple and quite efficient but both makes messages a little larger than they need be, and means that this type of bridge interceptor can not be nested (i.e. the original message port can not itself be a bridge interceptor).

Because EEK both uses a single address space and only a single memory domain it is possible to use an internal signal binding directly from "outside" a thread. For example, a function-based signal binding can be exported from a process where the function could still be invoked. The effects will be:

- there will be no message passing or context switching latency,
- the signal function (used to deliver the message) will be called using the current processor stack and not the one belonging to the exporting thread[12]; and,
- the invocation of the signal function will not implicitly be synchronized with other processing in the exporting thread.

Where these features are disadvantageous (e.g. potentially requiring very large stacks in Interrupt Service Routines that cause signals) external bindings, created by a bridge interceptor, can be used. If the purpose of using signal bindings and signal ports is to achieve some level of independence from the computational object abstraction provided by an underlying kernel, external bindings (or, more generally, bindings appropriate to the invocation domain in which they are to be used) should always be generated.

8 Conclusions

A set of simple data structures capable of supporting the abstractions of signals, signal bindings, signal ports, events, traders and bridge interceptors has been described. Their design has been driven by a requirement to provide functionality that could extend to much more complex scenarios than would initially be anticipated in an embedded system, whilst limiting their "common case" implementation cost and complexity to that appropriate for embedded systems.

The sufficiency of the chosen structures to represent a computational model, as exemplified by the standard ODP reference model, has been outlined and this should give some confidence that they will be appropriate in other environments. Their implementation in EEK has lead to a kernel that supports a flexible and highly configurable set of interacting components.

9 References

[APM 93] Architecture Project Management Ltd, "The ANSA Model for Trading and Federation", AR.005.00, published by authors, Feburary 1993.

[Bennett et al 97] Bennett F, Clarke D, Evans J B, Hopper A, Jones A, Leask D, "Piconet – Embedded Mobile Networking", IEEE Personal Communications, Vol 4 No 5, October 1997, pp 8-15.

[Black 95] Black R J, "Explicit Network Scheduling", Technical Report No 361, University of Cambridge Computer Laboratory, April 1995.

[Hoare 85] Hoare C A R, "Communicating Sequential Processes", Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK)

[ISO 95] ISO/IEC JTC1/SC21, "Information technology – Open Distributed Processing – Reference Model", ITU-T X.901 to X.904 or ISO/IEC 10746:1995, May 1995.

[ISO 96] ISO/IEC JTC1/SC21, "Information technology – Open Distributed Processing – Trading Function", ITU-T Recommendation X.930 or ISO 13235:1996, February 1996.

[Labrosse 92] Labrosse J J, "µC/OS The Real-Time Kernel", R & D Publications Inc, Lawrence, Kansas, 1992.

[Lauer et al 78] Lauer H C, Needham R M, "On the Duality of Operating System Structures" Proceedings of the Second International Symposium on Operating Systems, IRIA, October 1978, reprinted in Operating Systems Review, Vol 13, No 2, April 1979, pp 3-19.

[12] If the internal binding incorporates a signal function that is simply a direct call to a message handling function this will involve the whole of the processing of the message.

[Needham et al 82] Needham R M, Herbert A J, “The Cambridge Distributed Computing System”, Addison Wesley, Reading MA, 1982.

[OMG 99] Object Management Group, “The Common Object Request Broker – Architecture and Specification” chapter 3 “IDL Syntax and Semantics”, revision 2.3.1, October 1999.

[Pountain 86] Pountain D, “Tripos - The Roots of AmigaDOS”, BYTE, Vol 11, No 2, pp 321-328, February 1986.

[Roscoe 95] Roscoe T, “The Structure of a Multi-Service Operating System”, Technical Report No 376, University of Cambridge Computer Laboratory, August 1995.

[van der Linden 93] van der Linden R J, “The ANSA Naming Model”, AR.003.01, Architecture Projects Management Ltd, February 1993.

[Virata 99] The ATMOS Book, Virata Ltd, Developers’ reference DO-007001-TC (TC0001), issue 8, 20 May 1999.

Index