Enhanced View of Time Specification

October 2004
Version 1.2
formal/04-10-04

OBJECT MANAGEMENT GROUP

An Available Specification of the Object Management Group, Inc.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Contents

Contents

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.
- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
  - *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
  - *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

Contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of the *Enhanced View of Time* specification:

- Altair Aerospace Corporation
- General Dynamics Information Systems
- Objective Interface Systems, Inc.

Overview 1

This specification is a new Clock Service that leaves the specification of Time Service unchanged.

Contents

This chapter contains the following sections.

1.1 Clocks

1.1.1 Definition

The term “clock,” as used in this document, is a logical entity that can yield a “time reading.” It is assumed that this reading in some way measures the passage of time. The relationship of the readings of a clock to physical time, if known, is characterized by a set of “clock characteristics.”

The existing *CORBA Services: Time Service Specification* recognized only one clock, one presumed to represent UTC (Universal Time Coordinated). While this clock is of primary importance for most applications, other applications require clocks with different characteristics. For example, applications may require clocks that:

- are strictly monotonic, constant rate. While UTC is constant rate, it is subject to the insertion of leap seconds. In some applications, a one second difference can cause an unacceptable error. For example, in satellite navigation, a one second error causes a seven kilometer error in position for a low-earth orbiting satellite.
- can be paused, continued, or reset. The countdown clock for the launch of the Space Shuttle may be the most well-known of this class of clocks.
- are relative to a certain event. “Mission time clocks” are of this flavor.

In addition, there are a set of clocks that are not coordinated with an external time source. These clocks, usually associated with some sort of local hardware oscillator, are often used because of the low latency of access to a local device, because a network is isolated from external sources, or because cost or size constraints prevent incorporation of software or hardware synchronization with external time sources.

In addition to the need for clocks with characteristics other than that provided by the existing Time Service, there is a need to recognize that multiple time sources are becoming available on many networks. Any network connected to the internet, given sufficient firewall support, has access to multiple external time sources. The presence of multiple external time sources on private networks is also becoming more common.

Conversely, there are often needs to access a time source that is not local. There are a number of embedded single-board computers where the only on-board clock has a resolution of 20 or 16 milliseconds (derived from a 50Hz or 60Hz power input). A CORBA call to a remote time source with a round-trip time of 500 microseconds can obviously increase the precision of any time or interval measurement.

This specification introduces a generalized **Clock** interface to represent clocks with differing characteristics. Each clock is capable of providing a readout of time and is characterized by a set of properties.

1.1.2 Characteristics

Clocks have a set of characteristics that may render them useful or useless in any particular application. Several of the characteristics that are applicable to any clock include:

- *resolution*: the granularity of readout of a clock. Also, the time interval during which the readout of a clock will not change. The resolution is usually the inverse of the oscillator driving the clock device.
- *precision:* the number of bits provided in the clock readout and their scaling. Usually, this is more bits than that required by the resolution of the clock. Therefore, the *resolution* of a clock is more often of significance to an application. However, all clocks will *roll-over*; that is, transition from a large number to zero. In

some applications, such as using time stamps to ensure uniqueness the time between roll-overs is important. This is determined by the resolution and precision of the clock.

- *stability:* the ability of a clock to report consistent intervals of time; that is, to "tick" at a constant rate. Stability is measured by some (small) number of derivatives of the clock rate, either overall (for example, aging of a crystal oscillator) or against environmental factors (for example, temperature).

While these characteristics are inherent in any clock, they can only be determined by measurement against an accepted standard time source. For many systems, the characterization of clocks will be limited to off-line, static measurements, or manufacturers specifications. In this specification, these clocks are termed *uncoordinated*.

When more than one clock is present in a system, a number of time-dependent pair-wise characteristics are relevant[1]:

- *offset*: the difference between two clocks at a particular instant in time. To allow direct support of clocks supporting local or mission time, offset will be subdivided into *deliberate offset* and *unsynchronized offset*.
- *skew*: the rate of change (first derivative) of the offset between two clocks (at a particular instant of time. Also, the difference in frequency of two clocks. To allow characterization of clocks that are rate adjusted to compensate for synchronization errors and to support clocks for certain types of simulation, this parameter will be subdivided into *deliberate skew* and *accidental skew*. To allow support of clocks that may pause and or reset during an interval, a special indication will be reported when a clock is or has been paused or has been reset during a measurement interval.
- *drift*: the rate of change of skew (second derivative of offset) between two clocks. A special indication will be defined if the deliberate skew has changed in a measurement interval.

When a clock can be compared against a clock that is accepted as a standard, or is accepted as synchronized with a standard, the *accuracy* of a clock can be characterized.

A number of network protocols have been included to allow physical clock sources to be adjusted, so that the resulting logical clocks appear synchronized with other clocks. In particular, NTP allows synchronization with primary, externally-driven time servers through hierarchically organized strata of secondary and peer time servers.

Clocks that are synchronized through NTP, other software protocols, or hardware means to another clock will be termed *coordinated clocks* in this specification. Coordinated clocks have additional characteristics that identify and characterize the

1. As characterized in RFC 1305, "Network Time Protocol (Version 3) Specification, Implementation, and Analysis", IETF

synchronization source. Unfortunately, these characteristics tend to be specific to the synchronization protocol. This specification includes the following clock characteristics for all coordinated clocks:

- *coordination time scale*: the time scale directly (through an external time source) or indirectly coordinated with. Usually UTC, but other members of the Universal Time family, and *local time* (for example, UTC offset for time zone and daylight time) are also used.
- *coordination strata*: an indication of “directness” of the coordination with the ultimate time source, usually an external hardware time source.
- *coordination source*: the source of coordination.

This specification includes a set of data structures for these characteristics and means to query for the characteristics of a clock. Querying is supported by the **ClockCatalog** interface.

1.1.3 Cataloging and Bootstrapping

The present Time Service recognizes only one time scale, UTC, and is silent on bootstrapping. In particular, there is no portable method to obtain a **TimeService** object reference.

This specification includes the provision for multiple clocks registered in a catalog and includes reserving additional **ObjectId**s for use in the **resolve_initial_references** call to allow portable bootstrapping.

The **ClockCatalog** is a specialized repository, it holds registrations for clocks and the known characteristics of those clocks. The catalog may be queried for the known characteristics of a clock. The **ClockCatalog** also supports registration and querying by name. This allows an application with full knowledge of its system context to almost directly obtain a known clock, while allowing other applications to select a clock based on the desired characteristics of a clock.

This specification includes the reservation of two additional **ObjectId**s for use in the **resolve_initial_references** operation. “**ClockService**” would return a reference to the **ClockCatalog**. “**LocalClock**” would return a reference to a clock object that reads the (coordinated or uncoordinated) local system clock, if any.

1.2 CosTime Service Reprised

The features of the present **CosTime** service are provided in a more usable manner by two value types (**UTC** and **TimeSpan**) and a specialized clock interface (**TimeService**) that yields readouts in the **TimeBase::UtcT** type. The **UTC** valuetype roughly replaces the **UTO** interface from **CosTime**, while the **TimeSpan** value type replaces the **TIO** interface. Neither of these interfaces in **CosTime** were meant to be used remotely. Indeed there is an admonition in the present specification that users should use instances of **UtcT** instead of instances of UTO in operation parameter lists.

The **UTO** and **TIO** interfaces were created to provide standard operations on **TimeBase::UtcT** and **TimeBase::IntervalT**. With the adoption of value types in the *CORBA/IIOP Specification*, these operations can now be defined on a construct that will be passed by value across the network.

The **TimeService** is very similar to that defined in **CosTime**. However, instead of returning references to instances of the **UTO** and **TIO** interfaces, the new value types are returned.

This specification presents cleaner, lighter weight interfaces to achieve the function of **CosTime**. However, this specification does not deprecate or otherwise change **CosTime**.

1.3 Synchronization

This specification includes interfaces to synchronize a Clock with a “master clock.” A master clock is one whose readings are “trusted” to be accurate enough for use in the application, either because the inherent accuracy and stability of the hardware source of time or because the master is itself synchronized to another master clock. Pairwise synchronization with a master clock is referred to as “external clock synchronization” in the literature[2].

Synchronization of a clock with a master clock requires two steps:

1. Determine the difference between the clocks. Note that while this can be as simple a process as reading the master clock, it may have to be repeated several times to minimize errors, ensure success, or build an adequate history to determine skew and drift.

2. Apply a correction to the raw output of the slaved clock source before presenting the clock reading to an application.

This process might best be done semi-autonomously since it is relatively long-running and must be periodically repeated to preserve application-specified or default bounds on errors.[3] However, this may require the dedication of a thread, and could introduce uncertainty into a real-time system. For this reason, the interfaces allow explicit control of “synchronization episodes” as well as transparent, semi-autonomous synchronization.

Inclusion of the synchronization requirements in the RFP was not without controversy. Note two things, however:

2. If master/slave synchronization is not sufficient, both the interaction protocol and the algorithms employed are more complex. See, Christian, F. and Christof Fetzer, “Probabilistic Internal Clock Synchronization”, Proceedings of the Thirteenth Symposium on Reliable Distributed Systems, Oct 1994, Dana Point, CA.

1. The ability to perform the functions in step 1 are separately and independently required by the RFP.
2. No special interoperability interfaces are required; the requirements on the master clock interface is limited to reading the remote clock.

This specification discusses coupling the synchronization requirements with the requirements to characterize the differences in the clocks. In particular, the derivatives of offset between two clocks will only be available for clocks that are coordinated and for which active synchronization has been requested.

Three interfaces support clock synchronization.

The **SynchronizeBase** adds one operation to the **Clock** interface. It requires a clock to be able to measure the interval in which it takes to obtain the time from a remote (presumably a master) clock. The length of this interval determines the accuracy to which a clock can be synchronized to the master. This interface is mainly provided as a building block for applications that implement a specialized synchronization algorithm.

Two additional interfaces are provided for synchronization: the **Synchronizable** interface is a factory interface that creates instances of the **SynchronizedClock** interface. The **new_slave** operation initiates active determination of the difference between a slave clock and its master and application of a correction to the slave. These clocks smoothly converge a clock with another; that is, its master. The operation parameters include setting error bounds and retry limits that can be used to control the periodicity of synchronization polling with the designated master.

The **SynchronizedClock** interface supports periodic updates of the synchronization information. It also provides for synchronization to be controlled through explicit requests to resynchronize a previously synchronized clock.

1.4 Controllable Clocks

Certain clocks can be paused and resumed, reset, or otherwise controlled. Examples include “mission clocks” and the clock controlling (American) football games. This specialized class of clocks is provided by the **ControlledClock** interface. This interface provides user controls to start, stop, set, or vary the rate of a clock.

1.5 Delayed Execution

No special interfaces are proposed for delayed execution. Delayed execution can be done by:

---

3. See, for example, Lamport, L. and P. M. Melliar-Smith, “Synchronizing Clocks in the Presence of Faults,” *Journal of the ACM*, Vol. 32, No. 1, January 1985, pp. 52-78 and Christian, F., “Probabilistic Clock Synchronization,” *Distributed Computing*, No. 3, 1989, pp. 146-158.

1. Converting the desired time in the specified view of time to UTC and using the **RequestStartTime** policy or **ReplyStartTime** policy as specified in the *CORBA Messaging Specification*. This may not account for discontinuity the time kept by a particular clock, especially for clocks that may be paused and/or reset.
   or

2. Using the period invocation interface, described below, and specifying an execution count of 1.

1.6 Periodic Execution

Certain operations, especially in Real-Time systems, will be executed periodically. While it is possible for users to perform periodic processing using operating system or language-supplied threading capabilities, it is not always possible to tie periodic processing to a particular clock, especially a remote one. This specification includes a **PeriodicExecution** interface. A **PeriodicExecution::Controller** reference can be obtained from an instance of the **PeriodicExecution::Executor** interface, a specialized **Clock** interface, by providing a reference to an instance of an object derived from the conceptually abstract **Periodic** interface. The **Controller** interface provides controls on periodic execution. An execution limit, a single type **any** data parameter, and time offsets may be provided when the **PeriodicExecution** is initiated. Other operations on the **PeriodicExecution** allow suspension, resuming, and termination of the periodic execution.

When enabled, the **Controller** will invoke the **do_work** operation on the specified object. This specification makes no provision for detecting or handling overruns.

Clock Service 2

This chapter defines the CORBA Clock Service. The Clock Service includes much of the functionality of the Time Service, along with enhancements to deal with multiple clocks, synchronization, and periodic execution. As a result, the requirements of the RFP for the Time Service were considered in addition to the requirements of the RFP for the Enhanced View of Time.

Contents

This chapter contains the following topics.

2.1 Introduction

2.1.1 Representation of Time

Time is represented many ways in programs. For example the *X/Open DCE Time Servic*e [1] defines three binary representations of absolute time, while the UNIX SVID defines a different representation of time. Other systems use time represented in myriads of different ways.

In order to remain compatible with the Time Service, the Clock Service generalizes the representation of time in a compatible way and offers facilities that use the single representation of time used by the Time Service (and in aspects of the *CORBA/IIOP Specification*, such as *CORBA Messaging*.)

The Clock Service uses the **TimeBase::TimeT** type as the readout type for all clocks. It also retains the time scale definition for the **TimeT** type:

Time units	100 nanoseconds (10^{-7} seconds)
Base time	15 October 1582 00:00:00.
Approximate range	AD 30,000

The corresponding binary representations of relative time is the same one as for absolute time, and hence with similar characteristics:

Time units	100 nanoseconds (10^{-7} seconds)
Approximate range	+/- 30,000 years

2.1.2 Sources of Time

The Clock Service depends only on sources of time that provide a signal or readout that corresponds, in some statistically characterizable way, to the passage of time. Each source of time is assumed to have some, possibly indirect, hardware support for the marking of the passage of time. This is true of clocks that are direct readouts of hardware time sources or clocks that are based on software smoothing, adjustment or other manipulation of a hardware signal.

Some sources are trusted[1] to be accurate so that they can be used as master clocks to which the inaccuracy of other clocks may be measured. Such “external clocks” are usually synchronized to some hardware source (GPS, WWV, etc.) of an accepted time base, such as UTC. In contrast, “internal clocks” are supported by some hardware, typically a non-temperature-compensated oscillator, and are not known to be accurate.

The Clock Service makes no assumption about the accuracy of underlying time sources. It provides, however, means for characterizing the properties of each available time source, so that applications may select among them. It also provides facilities for

1.Not necessarily in the security sense.

requesting the creation of a new clock, tied to a designated internal clock for real-time timing information, but synchronized to a designated external clock within some accuracy and probability bounds.

2.1.3 General Object Model

The object model for the Clock Service supports multiple time sources. The source of time measurements is a **Clock** interface. The base **Clock** interface has an attribute that lets the applications examine the properties of the clock and select among different time sources in that way. The selection of clocks is further supported by a **ClockCatalog** interface that serves as a registry for clocks.

Specializations of the **Clock** interface include:

- **TimeService** interface - supports readouts of the **Timebase::UtcT** type supported by the Time Service. However, the readout is returned in a new **UTC** value type, instead of the “wrapper object” used by the Time Service.
- **SynchronizeBase** interface - a building block interface useful for building developer-defined conversion or synchronization facilities.
- **Synchronizable** interface - allows the creation of a virtual clock, an instance of the **SychronizedClock** interface, that presents a view of the clock corrected to synchronize with a designated master within a prescribed error bounds.
- **SychronizedClock** interface - a view of clock that is corrected to synchronize with a designated master clock.
- **ControlledClock** interface - a clock with operations that allow it to be paused, reset, etc.
- **PeriodicExecution::Executor** interface - supports active periodic execution of a specified method of an object. This interface returns an instance of the **PeriodicExecution::Controller** interface when an object derived from the **PeriodExecution::Periodic** interface is registered. The **Controller** object allows control over the periodic execution.

2.2 TimeBase Module

The Clock Service reuses the data structures in the **TimeBase** module. The **TimeBase** module was defined separately so that other services can make use of these data structures without requiring the interface definitions from either the Time Service or the Clock Service. The definitions of the **TimeBase** module are repeated here for completeness. They are not a normative part of this specification, since they are defined elsewhere.

2.2.1 Data Types

A number of types and interfaces are defined and used by this service. Most definitions of data structures are placed in the **TimeBase** module. All interfaces, and associated enum and exception declarations are placed in the **CosClockService** module. This separation of basic data type definitions from interface-related definitions allows other services to use the time data types without explicitly incorporating the interfaces, while allowing clients of those services to use the interfaces provided by the Clock Service to manipulate the data used by those services.

```
// IDL
module TimeBase {
    typedef unsigned long long  TimeT;
    typedef TimeT               InaccuracyT;
    typedef short               TdfT;
    struct UtcT {
        TimeT                   time;           // 8 octets
        unsigned long           inacclo;        // 4 octets
        unsigned short          inacchi;        // 2 octets
        TdfT                    tdf;            // 2 octets
                                                // total 16 octets.
    };
        struct IntervalT {
            TimeT               lower_bound;
            TimeT               upper_bound;
    };
};
```

2.2.1.1 Type TimeT

TimeT represents a single time value, which is 64 bits in size, and holds the number of 100 nanoseconds that have passed since the base time. For absolute time the base is 15 October 1582 00:00 of the Gregorian Calendar. All absolute time shall be computed using dates from the Gregorian Calendar.

2.2.1.2 Type InaccuracyT

InaccuracyT represents the value of inaccuracy in time in units of 100 nanoseconds. As per the definition of the inaccuracy field in the *X/Open DCE Time Service* [1], 48 bits is sufficient to hold this value.

2.2.1.3 Type TdfT

TdfT is of size 16 bits short type and holds the time displacement factor in the form of minutes of displacement from the Greenwich Meridian. Displacements East of the meridian are positive, while those to the West are negative.

2.2.1.4 Type UtcT

UtcT defines the structure of the time value that is used universally in this service. The basic value of time is of type **TimeT** that is held in the time field. Whether a **UtcT** structure is holding a relative time (that is, a duration) or an absolute time is determined by context; there is no explicit flag within the object holding that state information. (Note that, if a **UtcT** structure is used to hold a duration, its **tdf** must be set to zero.)

The **iacclo** and **inacchi** fields together hold a 48-bit estimate of inaccuracy in the time field. These two fields together hold a value of type **InaccuracyT** packed into 48 bits. The **tdf** field holds time zone information. Implementations must place the time displacement factor for the local time zone in this field whenever they create a **UTO** that expresses absolute time.

The time field of a **UtcT** used to express absolute time holds **UTC** time, irrespective of the local time zone. For example, to express the time 3:00pm in Germany (which is one hour east of the Universal Time Zone), the time field must be set to 2:00pm on the given date, and the **tdf** field must be set to 60. This means that, for any given **UtcT** value '**utc**', the local time can be computed as

utc.time + utc.tdf * 600,000,000

Note that it is possible to produce correct **UtcT** values by always setting the **tdf** field to zero and only setting the time field to **UTC** time; however, implementations are encouraged to include the local time zone information for the **UtcT** values they produce.

2.2.1.5 Type IntervalT

This type holds a time interval represented as two **TimeT** values corresponding to the lower and upper bound of the interval. An **IntervalT** structure containing a lower bound greater than the upper bound is invalid. For the interval to be meaningful, the time base used for the lower and upper bound must be the same, and the time base itself must not be spanned by the interval.

2.3 CosClockService Module

The remaining IDL definitions are contained in the new **CosClockService** module.

2.4 Clocks

2.4.1 Properties of Clocks

The following module supports the characterization of clocks:

```
// IDL
module CosClockService
{

    interface Clock;

    module ClockProperty
    {     // the minimum set of properties to be supported for a clock

          typedef unsigned long Resolution; // units = nanoseconds
          typedef short Precision;  // ceiling of log_2(seconds signified by least
                                    // significant bit of time readout)
```

```
typedef unsigned short Width; // no. of bits in readout - usually <= 64
typedef string Stability_Description;

typedef short Coordination;
const Coordination Uncoordinated = 0; // only static characterization
                                      // is available
const Coordination Coordinated   = 1; // measured against another
                                      // source
const Coordination Faulty= 2;         // e.g., there is a bit stuck

// the following are only applicable for coordinated clocks
struct Offset
{
    long long measured;     // units = 100 nanoseconds
    long long deliberate;   // units = 100 nanoseconds
};


typedef short Measurement;
const Measurement Not_Determined = 0;  // has not been measured
const Measurement Discontinuous  = 1;  // e.g., one clock is paused
const Measurement Available= 2;        // has been measured

typedef float Hz;
struct Skew
{
    Measurement available;
    Hz measured; // only meaningful if available = Available - in Hz
    Hz deliberate; // in Hz
};
typedef float HzPerSec;
struct Drift
{
    Measurement available;
    HzPerSec measured;      // meaningful if available = Available
                            // in Hz/sec
     HzPerSec deliberate;       // in Hz/sec
};

typedef short TimeScale;
const TimeScale Unknown = -1;
const TimeScale TAI      = 0;     // International Atomic Time
const TimeScale UT0      = 1;     // diurnal day
const TimeScale UT1      = 2;     // + polar wander
const TimeScale UTC      = 3;     // TAI + leap seconds
const TimeScale TT       = 4;     // terrestrial time
const TimeScale TDB      = 5;     // Barycentric Dynamical Time
const TimeScale TCG      = 6;     // Geocentric Coordinate Time
const TimeScale TCB      = 7;     // Barycentric Coordinate Time
const TimeScale Sidereal = 8;     // hour angle of vernal equinox
const TimeScale Local    = 9;     // UTC + time zone
const TimeScale GPS      = 10;    // Global Positioning System
const TimeScale Other    = 0x7fff; // e.g. mission

typedef short Stratum;
```

```
        const Stratum unspecified = 0;
        const Stratum primary_reference = 1;
        const Stratum secondary_reference_base = 2;

        typedef Clock CoordinationSource; // what clock is coordinating with
        typedef string Comments;
    };
```

These properties may be measured or set at configuration time for the known clocks. Note that they are cataloged as properties, thus they may be suitable for use in a Trader Service.

2.4.2 The Clock Interface

The **Clock** interface is the base interface for all clocks. It has the following definition:

```
// IDL
module CosClockService
{
    exception TimeUnavailable {};

    // the basic clock interface
    interface Clock // a source of time readings
    {
        readonly attribute CosPropertyService::PropertySet properties;
        readonly attribute TimeBase::TimeT current_time
            getRaises(TimeUnavailable);
    };
```

2.4.2.1 Exception TimeUnavailable

This exception is raised whenever the underlying clock fails, or is unable to provide time that meets the required security assurance.

2.4.2.2 Readonly attribute properties

The known properties of the clock.

2.4.2.3 Readonly attribute current_time

The clock’s current measurement of time.

2.5 UTC TimeService

This service replaces the CORBA Time Service.

2.5.1 Object Model

The **UTC** value type provides operations on the **TimeBase::UtcT** structure. These operations include comparisons with other instances, with and without consideration of the accuracy of the times being compared. The **UTC** value type replaces the **UTO** interface from the Time Service.

The **TimeSpan** value type provides operations on the **TimeBase::IntervalT** structure. These operations include determination of spans and overlaps between **TimeSpan**s and **UtcT**s. The **TimeSpan** value type replaces the **TIO** interface from the Time Service.

The **UtcTimeService** interface creates **UTC** value types that represent the time at which they were created. This interface replaces the **TimeService** interface from the Time Service.

2.5.2 Data Types

```
// IDL
module CosClockService
{
    enum TimeComparison
    {
        TCEqualTo,
        TCLessThan,
        TCGreaterThan,
        TCIndeterminate
    };

    enum ComparisonType
    {
        IntervalC,
        MidC
    };

    enum OverlapType
    {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
    };
```

2.5.2.1 Enum ComparisonType

ComparisonType defines the two types of time comparison that are supported. **IntervalC** comparison does the comparison taking into account the error envelope. **MidC** comparison just compares the base times. A **MidC** comparison can never return **TCIndeterminate**.

2.5.2.2 Enum TimeComparison

TimeComparison defines the possible values that can be returned as a result of comparing two UTCs. The values are self-explanatory. In an **IntervalC** comparison, **TCIndeterminate** value is returned if the error envelopes around the two times being compared overlap. For this purpose the error envelope is assumed to be symmetrically placed around the base time covering time-inaccuracy to time+inaccuracy. For **IntervalC** comparison, two **UTC**s are deemed to contain the same time only if the **Time** attribute of the two objects are equal and the **Inaccuracy** attributes of both the objects are zero.

2.5.2.3 Enum OverlapType

OverlapType specifies the type of overlap between two time intervals. Figure 2-1 depicts the meaning of the four values of this enum. When interval A wholly contains interval B, then it is an OTContainer of interval B and the overlap interval is the same as the interval B. When interval B wholly contains interval A, then interval A is OTContained in interval B and the overlap region is the same as interval A. When neither interval is wholly contained in the other but they overlap, then the **OTOverlap** case applies and the overlap region is the length of interval that overlaps. Finally, when the two intervals do not overlap, the **OTNoOverlap** case applies.

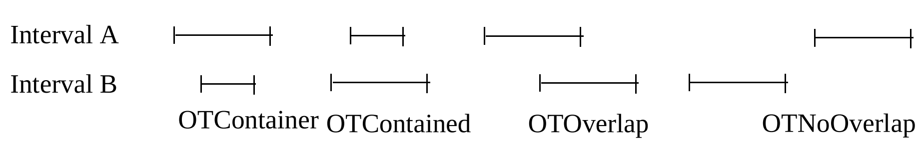


Figure 2-1 Illustration of Interval Overlap

2.5.3 Universal Time Coordinated (UTC)

The **UTC** value type provides various operations on basic time. These include the following groups of operations:

- Construction of a UTC from piece parts, and extraction of piece parts from a UTC (as read only attributes).
- Comparison of time.
- Conversion from relative to absolute time, and conversion to an interval.

```
// IDL
module CosClockService {
    valuetype TimeSpan;

    // replaces UTO from CosTime
    valuetype UTC
    {
```

```
    factory init(in TimeBase::UtcT from);
    factory compose(    in TimeBase::TimeT       time,
                        in unsigned long         inacclo,
                        in unsigned short        inacchi,
                        in TimeBase::TdfT        tdf);
    public TimeBase::TimeT       time;
    public unsigned long         inacclo;
    public unsigned short        inacchi;
    public TimeBase::TdfT        tdf;
    TimeBase::InaccuracyT        inaccuracy();
    TimeBase::UtcT               utc_time();
    TimeComparison compare_time(in ComparisonType comparison_type,
                                 in UTC with_utc);
    TimeSpan interval();
};
```

2.5.3.1 *Factory init*

Creates a **UTC** from a **TimeBase::UtcT**.

2.5.3.2 *Factory compose*

Composes a UTC from its piece parts.

2.5.3.3 *Public state member time*

Corresponds to the time member of the **UtcT** struct.

2.5.3.4 *Public state member inacclo*

Corresponds to the **inacclo** member of the **UtcT** struct.

2.5.3.5 *Public state member inacchi*

Corresponds to the **inacchi** member of the **UtcT** struct.

2.5.3.6 *Public state member tdf*

Corresponds to the **tdf** member of the **UtcT** struct.

2.5.3.7 *Operation inaccuracy*

This is the inaccuracy attribute of a UTO represented as a value of type **InaccuracyT**.

2.5.3.8 *Operation utc_time*

This is the time expressed as a **TimeBase::UtcT** type.

2.5.3.9 Operation compare_time

Compares the time contained in the value with the time given in the input parameter **with_utc** using the comparison type specified in the **in** parameter **comparison_type**, and returns the result. See the description of **TimeComparison** in Section 2.5.2, “Data Types,” on page 2-8, for an explanation of the result. See the explanation of **ComparisonType** in Section 2.5.2, “Data Types for an explanation of comparison types. Note that the time in the value is always used as the first parameter in the comparison. The time in the **with_utc** parameter is used as the second parameter in the comparison.

2.5.3.10 Operation interval

Returns a **TimeSpan** value representing the error interval around the time value in the **UTC** as a time interval.

TimeSpan.upper_bound = UTC.time + UTC.inaccuracy.
TimeSpan.lower_bound = UTC.time - UTC.inaccuracy.

2.5.4 TimeSpan Value

A **TimeSpan** value represents a time interval and contains operations relevant to time intervals.

```
// IDL
module CosClockService
{
    // replaces TIO from CosTime
    valuetype TimeSpan
    {
        factory init   (in TimeBase::IntervalT from);
        factory compose(     in TimeBase::TimeT lower_bound,
                             in TimeBase::TimeT upper_bound);

        public TimeBase::TimeT lower_bound;
        public TimeBase::TimeT upper_bound;
        TimeBase::IntervalT time_interval();
        OverlapType spans (
                in  UTC                time,
                out TimeSpan           overlap);
        OverlapType overlaps (
                in TimeSpan            other,
                out TimeSpan           overlap);
        UTC time ();
    };
```

2.5.4.1 Factory init

Creates a **TimeSpan** from a **TimeBase::IntervalT**.

2.5.4.2 Factory compose

Composes a **TimeSpan** from an upper and lower bound.

2.5.4.3 Public state member lower_bound

The lower bound of the time span.

2.5.4.4 Public state member upper_bound

The upper bound of the time span.

2.5.4.5 Operation time_interval

This attribute returns an **IntervalT** structure with the values of its fields filled in with the corresponding values from the **TimeSpan**.

2.5.4.6 Operation spans

This operation returns a value of type **OverlapType** depending on how the interval in the object and the time range represented by the parameter **time** overlap. See the definition of **OverlapType** in Section 2.5.2, "Data Types," on page 2-8. The interval in the object is interval A and the interval in the parameter UTC is interval B. If **OverlapType** is not **OTNoOverlap**, then the **out** parameter **overlap** contains the overlap interval; otherwise, the **out** parameter contains the gap between the two intervals. The exception CORBA::BAD_PARAM is raised if the UTC passed in is invalid.

2.5.4.7 Operation overlaps

This operation returns a value of type **OverlapType** depending on how the interval in the object and interval in the parameter **other** overlap. See the definition of **OverlapType** in Section 2.5.2, "Data Types." The interval in the object is interval A and the interval in the parameter **other** is interval B. If **OverlapType** is not **OTNoOverlap**, then the **out** parameter **overlap** contains the overlap interval; otherwise, the **out** parameter contains the gap between the two intervals. The exception CORBA::BAD_PARAM is raised if the **TimeSpan** passed in is invalid.

2.5.4.8 Operation time

Returns a **UTC** in which the inaccuracy interval is equal to the time interval in the **TimeSpan** and time value is the midpoint of the interval.

2.5.5 UTC Time Service

The **UtcTimeService** interface provides operations for obtaining the current time.

```
// IDL
module CosClockService
{
    interface UtcTimeService : Clock
    {
        UTC universal_time() raises(TimeUnavailable);
        UTC secure_universal_time() raises(TimeUnavailable);
        UTC absolute_time(in UTC with_offset) raises(TimeUnavailable);
    };
```

2.5.5.1 Operation universal_time

The **universal_time** operation returns the current time and an estimate of inaccuracy in a **UTC**. It raises TimeUnavailable exceptions to indicate failure of an underlying time provider. The time returned in the **UTC** by this operation is not guaranteed to be secure or trusted. If any time is available at all, that time is returned by this operation.

2.5.5.2 Operation secure_universal_time

The **secure_universal_time** operation returns the current time in a **UTC** only if the time can be guaranteed to have been obtained securely. In order to make such a guarantee, the underlying Time Service must meet the criteria to be followed for secure time, presented in "Appendix B, Implementation Guidelines." If there is any uncertainty at all about meeting any aspect of these criteria, then this operation must return the TimeUnavailable exception. Thus, time obtained through this operation can always be trusted.

2.5.5.3 Operation absolute_time

The **absolute_time** operation returns a new **UTC** containing the absolute time corresponding to the present time offset by the parameter **with_offset**. Raises a CORBA::DATA_CONVERSION exception if the attempt to obtain an absolute time causes an overflow.

2.6 The Clock Catalog Interface

The **ClockCatalog** interface allows applications to discover and select a clock for use. It is intended to be a light-weight alternative to the use of the Trading Service (for example, in embedded systems). It has the following definition:

```
// IDL
module CosClockService
{
    interface ClockCatalog {

        struct ClockEntry {
            Clock           subject;
            string          name;
        };
         typedef sequence<ClockEntry> ClockEntries;
```

```
        exception UnknownEntry {};

        ClockEntry get_entry(in string with_name) raises (UnknownEntry);
        ClockEntries available_entries();

        void register(in ClockEntry entry);
        void delete_entry(in string with_name) raises (UnknownEntry);
    };
```

2.6.1 Struct ClockEntry

This structure holds the known information about a clock: its registered name and its object reference.

2.6.2 Exception UnknownEntry

Indicates that the catalog contains no entry with the given name.

2.6.3 Operation get_entry

Retrieve the information know about a clock, given its registered name.

2.6.4 Operation available_entries

Retrieve the entire catalog so that the client may select a clock based on its known properties.

2.6.5 Operation register

Register a new clock with the catalog.

2.6.6 Operation delete_entry

Remove an entry from the registry.

2.7 Mission Time

Certain clocks, such as those used to time an (American) football game, may track the elapsed time from an event, and may need to be paused and resumed, and may need to be occasionally reset. The **ControlledClock** interface provides a specialization of the **Clock** interface with these controls. It has the following definition:

```
// IDL
module CosClockService
{
    // a controllable clock
    interface ControlledClock: Clock
```

```
{
    exception NotSupported {};
    void set(in TimeBase::TimeT to)    raises (NotSupported);
    void set_rate(in float ratio)      raises (NotSupported);
    void pause()                       raises (NotSupported);
    void resume()                      raises (NotSupported);
    void terminate()                   raises (NotSupported);
};
```

2.7.1 Exception NotSupported

The NotSupported exception may be raised if the operation is not supported for the instance of the **ControlledClock**, or if its characteristics disallow the operation. For example, the rate of a “mission clock” may not be settable. Other clocks may not be allowed to run “backwards.”

2.7.2 Operation set

Sets the current time maintained by the clock to the value specified.

2.7.3 Operation set_rate

Allows a clock to be speeded up or slowed down (or run backwards). The parameter indicates the ratio of the elapse of the clock’s readout to the real passage of time.

2.7.4 Operation pause

Pause the apparent elapse of time.

2.7.5 Operation resume

Resume the elapse of time.

2.7.6 Operation terminate

Stop the clock.

2.8 Synchronization

Three interfaces are defined to support synchronization of a clock with a master.

2.8.1 SynchronizeBase Interface

The **SynchronizeBase** interface adds a primitive operation to the **Clock** interface that allows the determination of an offset between two clocks and the error in that determination. It has the following definition:

```
// IDL
module CosClockService
{
    interface SynchronizeBase : Clock
    {
        struct SyncReading
        {
            TimeBase::TimeT local_send;
            TimeBase::TimeT local_receive;
            TimeBase::TimeT remote_reading;
        };
        SyncReading synchronize_poll(in Clock with_master);
    };
```

2.8.1.1 Struct SyncReading

A structure with three time components representing the local start and stop time of a query on another clock, and the reading corresponding that query.

2.8.1.2 Operation synchronize_poll

Instructs the clock to perform the following sequence of steps and return the result:

1. Place the clock's current reading into **local_send**.
2. Obtain the **with_master** clock's time; that is, invoke **readout** on it. Save it in **remote_reading**.
3. Place the clock's current reading into **local_receive**.

These steps should be performed with as little latency as possible. For example, possibly storage of values in the output structure should be delayed until all readings have been obtained. The goal is to decrease the interval between **local_send** and **local_receive**, since it represents twice the maximum error in an estimate of the offset between the clock and the designated master clock.

Clients of a clock can repeat this synchronization polling over time to obtain, for example, the frequency skew and drift between a clock and its master.

This operation times the round trip to read the **current_time** attribute of another clock. This bounds the offset between two clocks, and provides the primitive samples for external synchronization algorithms. For example, a single polling can yield an estimate of the clock offset as follows:

$$offset = \left(\left(remotereading - \frac{(localsend + localreceive)}{2}\right) \pm \frac{(localreceive - localsend)}{2}\right) \quad \textbf{(EQ 1)}$$

2.8.2 Synchronizable Interface

An instance of the **Synchronizable** interface allows the creation of new logical clock that relies on the synchronizable clock for a perception of the passage of time, but is adjusted to stay within a certain error bounds of another, presumably more accurate, "master" clock. This new clock is said to be synchronized, or slaved, to the master. The interface has the following definition:

```
// IDL
module CosClockService
{
    interface SynchronizedClock;

    exception UnableToSynchronize
    {
        TimeBase::InaccuracyT minimum_error;
    };

    interface Synchronizable : SynchronizeBase
    {
        const TimeBase::TimeT Forever = 0xFFFFFFFFFFFFFFFF;

        SynchronizedClock new_slave
            (in Clock                          to_master,
            in TimeBase::InaccuracyT           to_within,
            in short                           retry_limit,
            in TimeBase::TimeT                 minimum_delay_between_syncs,
            in CosPropertyService::Properties  properties
            ) raises (UnableToSynchronize);
    };

    interface SynchronizedClock : Clock
    {
        void resynch_now() raises (UnableToSynchronize);
    };
```

2.8.2.1 Exception UnableToSynchronize

This exception will be raised by the **new_slave** operation if the requested accuracy cannot be obtained after the prescribed number of retries. The exception will report the accuracy that was obtained.

2.8.2.2 Operation new_slave

Creates a new "slave" clock, an instance of the **SynchronizedClock** interface, that attempts to adjust the readings of the source clock to synchronize it **to_within** the specified error bounds. The **retry_limit** specifies the number of attempts to achieve the specified accuracy before an UnableToSynchronize exception can be raised. Once synchronized, the resulting **SynchronizedClock** instance must periodically re-read the master clock and resynchronize in order to maintain the specified level of accuracy. A conforming implementation must be able to do this autonomously. The

minimum_delay_between_syncs parameters specify a minimum period between these resynchronization episodes, thus allowing the number of remote readings of the master clock to be limited. Setting the **minimum_delay_between_syncs** parameter to the constant value **Forever** precludes the **SynchronizedClock** from autonomously resynching.

2.8.3 SynchronizedClock Interface

The **SynchronizedClock** interface provides a virtual clock that adjusts the readings of an underlying clock to be synchronized with a master. Instances are capable of determining the offset from a master by polling the time of the master and applying a synchronization algorithm to attain a specified accuracy with the master clock. Conforming implementations must be able to maintain the specified accuracy, usually by autonomously redetermining the offset from the master clock periodically. Instances of the **SynchronizedClock** interface are created by invoking the **new_slave** operation on an instance of the **Synchronizable** interface.

The interface is defined as follows:

```
// IDL
module CosClockService
{
    interface SynchronizedClock : Clock
    {
        void resynch_now() raises (UnableToSynchronize);
    };
```

2.8.3.1 Operation resynch_now

Instances of the **SynchronizedClock** interface may be precluded from autonomously initiating a series of readings of the master clock by specifying a **minimum_delay_between_syncs** of **Forever**. In this case, or if the application wishes maximum accuracy of the synchronization at a particular instant, the **resynch_now** operation will immediately resynchronize with the master clock.

2.9 Bootstrapping

To allow bootstrapping of applications, the following two **ObjectId**s are reserved for use in the **resolve_initial_references** operation:

1. Specifying “**TimeService**” yields a reference to a **ClockCatalog** object.
2. Specifying “**LocalClock**” yields a reference to the local system clock, if any.

2.10 PeriodExecution Service

Certain operations, especially in Real-Time systems, will be executed periodically. While it is possible for users to perform periodic processing using native or language-supplied threading capabilities, it is not always possible to tie periodic processing to a particular clock, especially a remote one. This service provides a useful and portable way to perform certain operations periodically. Three interfaces are defined in the **CosClockService::PeriodicExecution** module:

```
// IDL
module CosClockService
{
    module PeriodicExecution
    {

        interface Periodic
        {
            boolean do_work(in any params);
        };

        interface Controller
        {
            exception TimePast {};
            void start
                (in TimeBase::TimeT       period,
                in TimeBase::TimeT        with_offset,
                in unsigned long          execution_limit, // 0 = no limit
                in any                    params);
            void start_at
                (in TimeBase::TimeT       period,
                in TimeBase::TimeT        at_time,
                in unsigned long          execution_limit, // 0 = no limit
                in any                    params) raises (TimePast);
            void pause();
            void resume();
            void resume_at(in TimeBase::TimeT at_time) raises(TimePast);
            void terminate();
            unsigned long executions();
        };

        interface Executor : Clock
        {
            Controller enable_periodic_execution(in Periodic on);
        };
    };
```

2.10.1 The Periodic Interface

Instances of objects that are to be periodically executed must be derived from the **Periodic** interface, implement a **do_work** operation, and have been activated on a POA.

2.10.1.1 Operation do_work

The **do_work** operation will be periodically invoked by this service. Each invocation will be passed the type **any** value registered by the **start** or **start_at** operations on the **Controller** instance. The user implementation of the **do_work** operation should return a value of **TRUE** to continue periodic invocation; a value of **FALSE** will terminate periodic invocation.

2.10.2 Controller Interface

Allows control of periodic execution after the appropriate object has been registered with the clock.

2.10.2.1 Exception time_past

Raised by the **start_at** or **resume_at** operations if the requested time is in the past.

2.10.2.2 Operation start

Initiates periodic execution with a specified period for a specified count of executions. Specifying an execution limit of 0 is interpreted as an unbounded number of executions. The **with_offset** parameter may be used to delay the start of the first execution. The value of the type **any** parameter **params** will be passed to each invocation.

2.10.2.3 Operation start_at

Identical to the **start** operation except that the **at_time** parameter specifies an absolute time for the start of the first execution.

2.10.2.4 Operation pause

Pauses periodic execution.

2.10.2.5 Operation resume

Resumes periodic execution.

2.10.2.6 Operation resume_at

Resumes periodic execution at a particular time.

2.10.2.7 Operation terminate

Terminates periodic execution.

2.10.2.8 Operation executions

Reports the number of executions that have already been initiated.

2.10.3 Interface Executor

Allows registration of an object reference with a clock capable of performing periodic execution.

2.10.3.1 Operation enable_periodic_execution

Register an instance of the **Periodic** interface for periodic execution.

Lightweight Time Service 3

Note – This chapter is based on the Lightweight Services specification (ptc/04-07-03).

Contents

This chapter contains the following topics.

3.1 Platform Independent Model

3.1.1 Overview

This section defines the Platform Independent Model (PIM) for the Lightweight Time Service. The Lightweight Time Service is intended to be a subset of the full CORBA Enhanced View of Time Service. The packages, interfaces, and classes appearing in this chapter are intended to model this subset and should map to the IDL for their counterparts in the CORBA Enhanced View of Time Service Specification (Version 1.1, May 2002). The descriptions of the interfaces, operations and their semantics are also intended to be identical to those defined by the CORBA Enhanced View of Time Service Specification (Version 1.1, May 2002) over this same subset.

3.1.2 Minor Conformance Points

The platform independent model of the Lightweight Time Service supports two *optional* minor conformance points: *Support of Multiple Clocks* and *Support of Periodic Execution Control*.

Support of Multiple Clocks

This conformance point controls the presence or absense of an *optional* model section. If the conformance point evaluates to true, the **ClockCatalog** interface and the **ClockEntry** structure are included in the model, providing support for multiple clocks.

Support of Periodic Execution Control

This conformance point controls the presence or absense of an optional model section. If the conformance point evaluates to true, the **PeriodicExecution** package is included in the model, thus providing support for clock-controlled periodic execution.

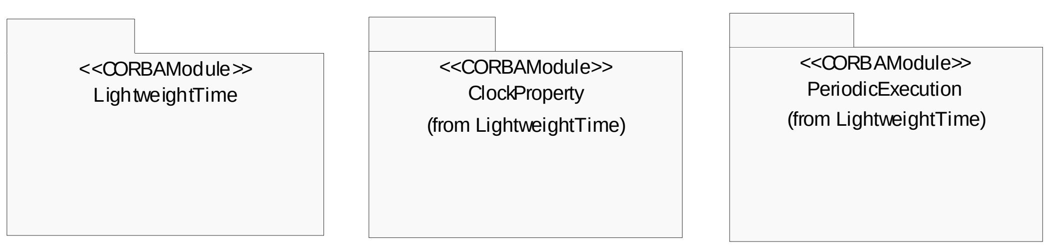


Figure 3-1 - Lightweight Time Service Package Structure

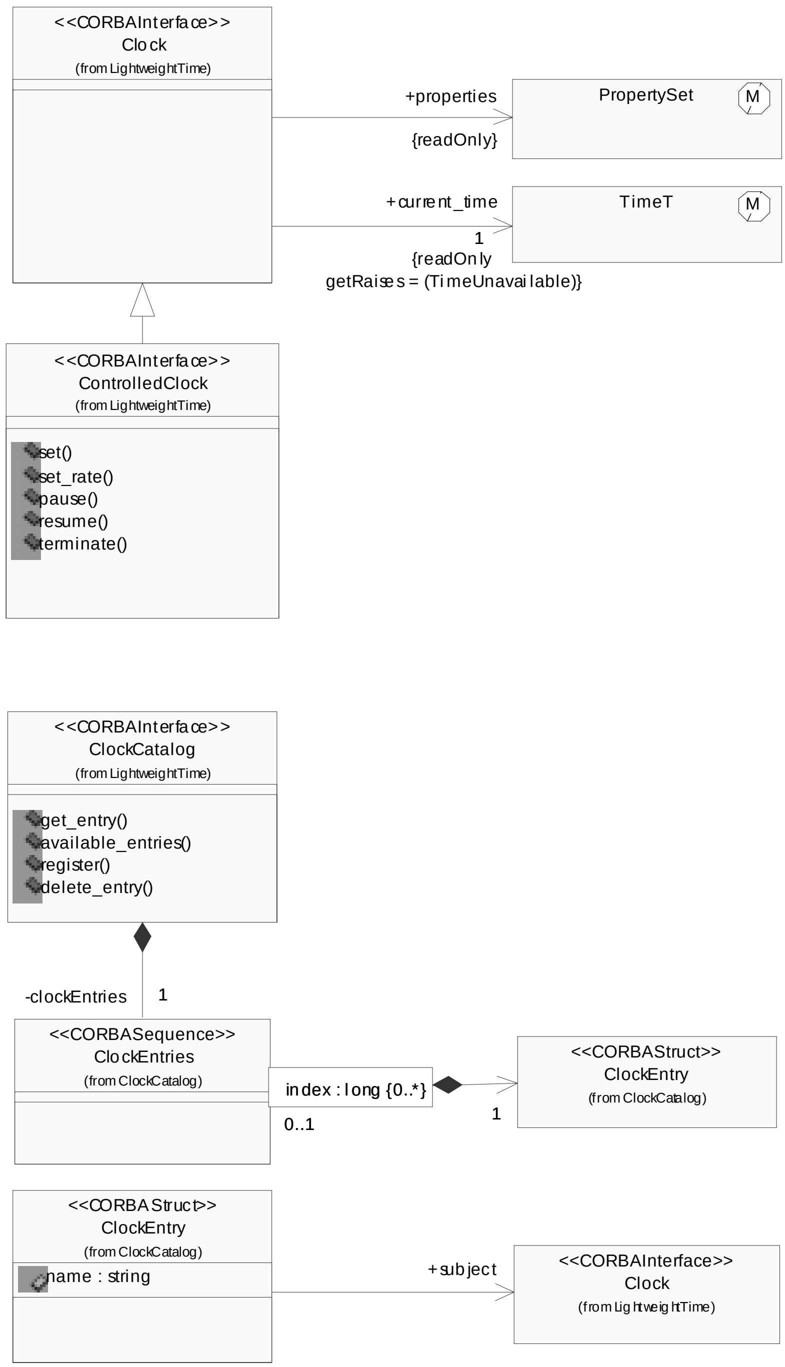
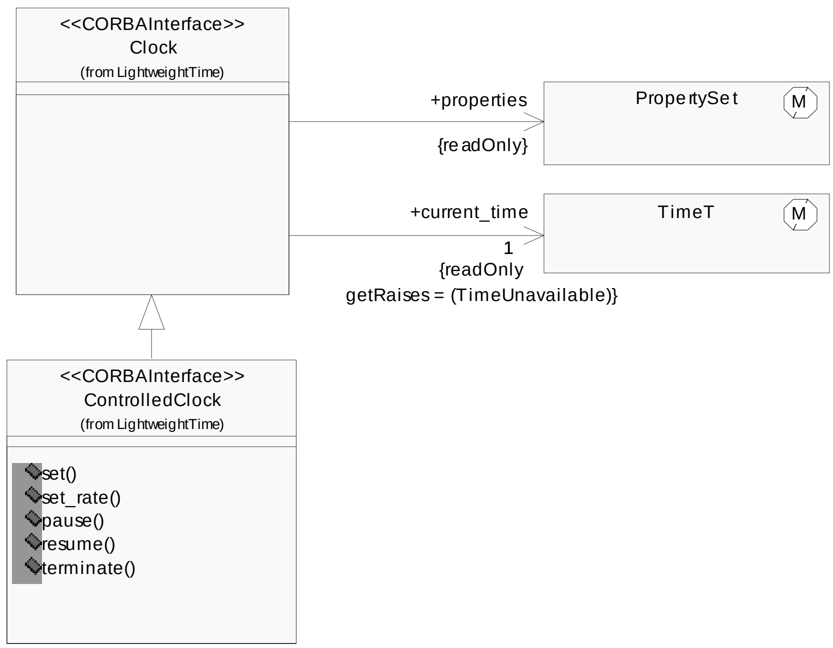


Figure 3-2 - Lightweight Time Service Interfaces and Classes

3.1.3 The LightweightTime Package

The **LightweightTime** package defines interfaces for finding a clock reading, a time source, controlling a clock and support for periodic execution. Synchronization of clocks is not supported in the **LightweightTime** package.



3.1.3.1 Clock

Description

Base interface for all clocks.

Attributes

No attributes.

Operations

No operations.

Associations

`properties: PropertySet [1]`

Points to a **PropertySet** holding the specific properties of the clock.

`current_time: TimeT [1]`

Points to a data element holding the current time as a 64-bit value with a resolution of 100 nanoseconds.

Constraints

No constraints.

Semantics

This is the base interface for all clocks defined in the Lightweight Time Service. It provides configurability for the clock via properties (name-value pairs) and access to a time base.

3.1.3.2 ControlledClock

Description

A user-controllable specialization of the **Clock** interface.

Attributes

No attributes.

Operations

`set(in t0: TimeT)`

This operation sets the controllable clock to the specified specific time.

`set_rate(in ratio: Float)`

This operation allows a clock to be speeded up or slowed down (or run backwards). The parameter indicates the ratio of the elapse of the clock’s readout to the real passage of time.

`pause()`

This opperation pauses the apparent elapse of time.

`resume()`

This operation resumes the apparent elapse of time.

`terminate()`

This operation stops the controlled clock permanently.

Associations

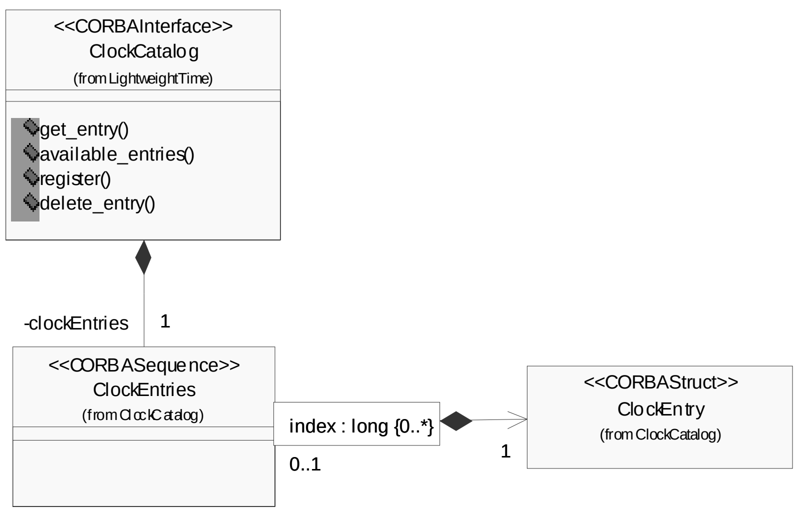
No additional associations.

Constraints

No Constraints.

Semantics

The **ControlledClock** is a specialization of the **Clock** interface. It provides the ability to set the clock to certain value, control the apparent “speed” (time elapse rate), and to pause and resume the clock under user control.



3.1.3.3 ClockCatalog

This interface is part of the optional minor conformance point “Support of Multiple Clocks.”

Description

A lightweight catalog of available clocks.

Attributes

No attributes.

Operations

`get_entry(in name: String): ClockEntry`

Returns a single clock entry holding the information about a particular clock. The clock entry is selected via the clock entry name.

`available_entries(): ClockEntries`

Returns the whole catalog to allow the client the application of a more specific selection mechaism, as for example by a specific property.

`register(in entry: ClockEntry)`

Register a new clock entry in the catalog.

`delete_entry()`

Permanently removes a clock entry from the clock catalog.

Associations

`clockEntries: ClockEntries[1]`

The encapsulation of the clock entry catalog content.

Constraints

No constraints.

Semantics

The **ClockCatalog** is the user-visible interface to a single-level lightweight trader service equivalent, holding information about available clock definitions.

3.1.3.4 ClockEntries

This set is part of the optional minor conformance point “Support of Multiple Clocks.”

Description

The set holding the individual clock entries.

Attributes

No attributes.

Operations

No operations.

Associations

`clockEntry: ClockEntry[*]`

The actual set holding the individual entries in the clock catalog.

Constraints

No constraints.

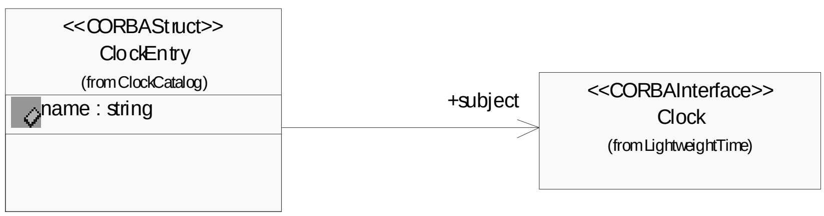
Semantics

Provides an encapsulation for the set of individual clock information entries.

3.1.3.5 *ClockEntry*

This interface is part of the optional minor conformance point “Support of Multiple Clocks.”

Description



An individual entry in the clock catalog.

Attributes

`name: String [1]`

The **ClockEntry** name.

Operations

No operations.

Associations

`clockl: Clock [1]`

The clock definition represented by this catalog entry.

Constraints

No constraints.

Semantics

A **ClockEntry** consists of a name (unique within the catalog) and a reference to a particular clock definition.

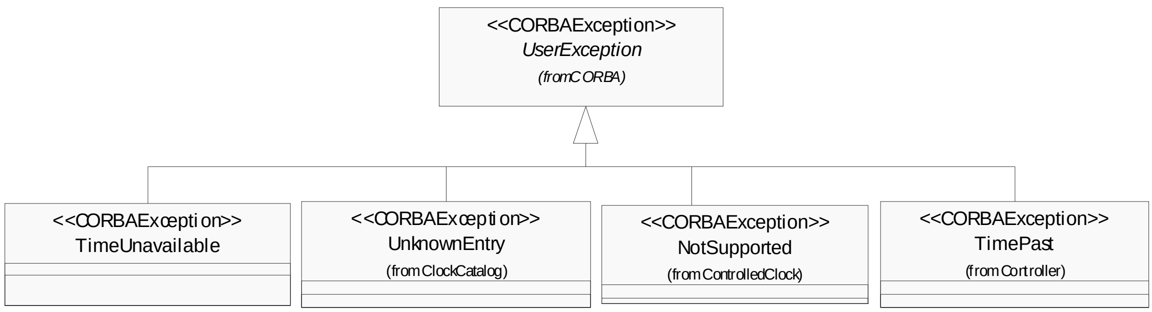


Figure 3-3 - Lightweight Time Service Exceptions

3.1.3.6 TimeUnavailable

Description

TimeUnavailable exception.

Attributes

No attributes.

Operations

No operations.

Associations

No associations.

Constraints

No constraints.

Semantics

This exception is raised whenever the underlying clock fails, or is unable to provide time that meets the required security assurance.

3.1.3.7 UnknownEntry

Description

UnknownEntry exception.

Attributes
No attributes.

Operations
No operations.

Associations
No associations.

Constraints
No constraints.

Semantics
Indicates that the catalog contains no entry with the given name.

3.1.3.8 NotSupported

Description
NotSupported exception.

Attributes
No attributes.

Operations
No operations.

Associations
No associations.

Constraints
No constraints.

Semantics
The NotSupported exception may be raised if the operation is not supported for the instance of the **ControlledClock**, or if its characteristics disallow the operation. For example, the rate of a **ControlledClock** may not be settable. Other clocks may not be allowed to run "backwards."

3.1.3.9 TimePast

Description

TimePast exception.

Attributes

No attributes.

Operations

No operations.

Associations

No associations.

Constraints

No constraints.

Semantics

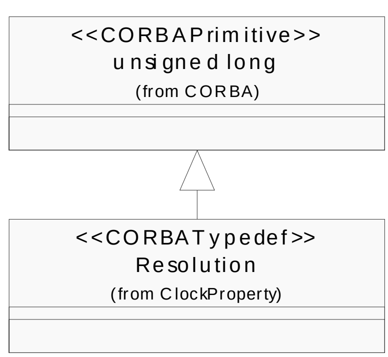
Raised by the **start_at** or **resume_at** operations if the requested time is in the past.

3.1.4 The ClockProperty Package

This package contains only data definitions. They constitute the minimum set of properties required for any clock.

3.1.4.1 Resolution

Description



Defines the apparent clock resolution.

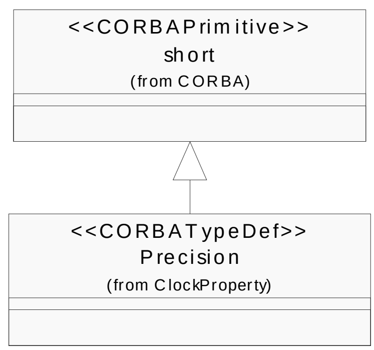
Constraints

Must be specified in units of nanoseconds.

Semantics

No special semantics.

3.1.4.2 Precision



Description

Defines the apparent clock precision.

Constraints

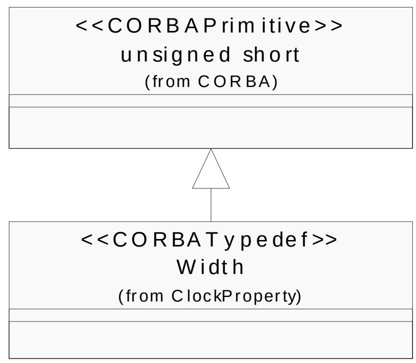
No constraints.

Semantics

Raised by the **start_at** or **resume_at** operations if the requested time is in the past.

3.1.4.3 Width

Description



Number of bits in clock readout.

Constraints

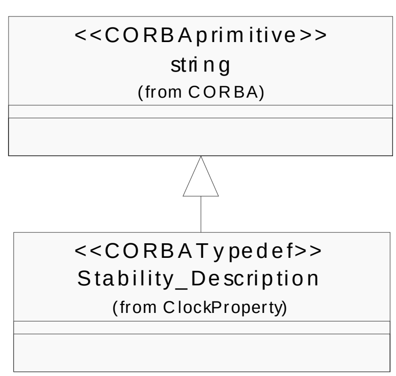
No constraints.

Semantics

Commonly used readout widths are less or equal 64 bits.

3.1.4.4 Stability_Description

Description



Describes the clock stability.

Constraints

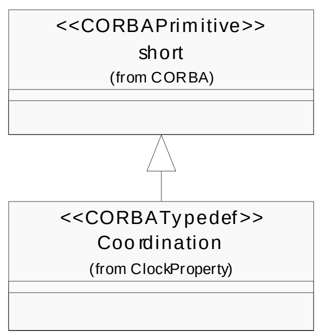
No constraints.

Semantics

No special semantics.

3.1.4.5 Coordination

Description



Defines the clock coordination method.

Constraints

Under the Lightweight Time Service, Coordination is restricted to the following set of values:

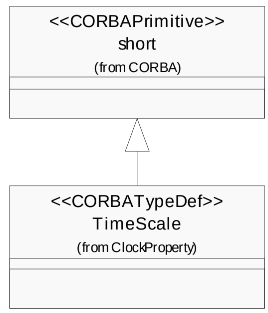
Name	Value	Meaning
`Uncoordinated`	`0`	only static characterization is available

Semantics

No special semantics.

3.1.4.6 TimeScale

Description



Defines the time scale used by the clock.

Constraints

Under the Lightweight Time Service, TimeScale is restricted to the following set of values:

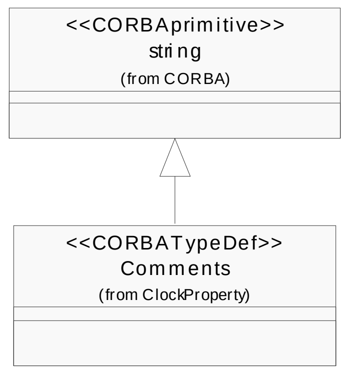
Name	Value	Meaning
`Unknown`	`-1`	
`TAI`	`0`	International Atomic Time
`UT0`	`1`	diurnal day
`UT1`	`2`	+ polar wander
`UTC`	`3`	TAI + leap second
`TT`	`4`	terrestrial time
`TDB`	`5`	Barycentric Dynamical Time
`TCG`	`6`	Geocentric Coordinated Time
`TCB`	`7`	Barycentric Coordinated Time
`Sidereal`	`8`	hour angle of veneral equinox
`Local`	`9`	UTC + time zone
`GPS`	`10`	Global Positioning System
`Other`	`0x7fff`	e.g., mission

Semantics

No special semantics.

3.1.4.7 Comments

Description



For supplemental comments.

Constraints

No constraints.

Semantics

No special semantics.

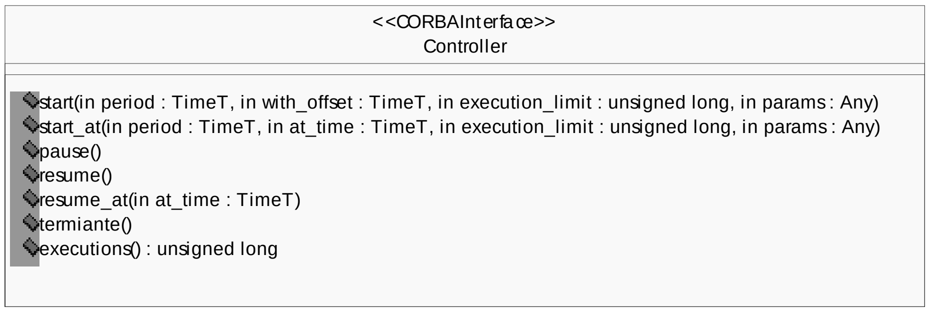
3.1.5 The PeriodicExecution Package

This package is part of the optional minor conformance point "Support of Periodic Execution Control."

3.1.5.1 Controller

This interface is part of the optional minor conformance point "Support of Periodic Execution Control."

Description



Controls periodic execution.

Attributes

No attributes.

Operations

`start(in period: TimeT, in with_offset: TimeT, in execution_limit: unsigned long, in params: Any)`

Initiates periodic execution with a specified period for a specified count of executions. Specifying an execution limit of 0 is interpreted as an unbounded number of executions. The **with_offset** parameter may be used to delay the start of the first execution. The value of the type any parameter params will be passed to each invocation.

`start_at(in period: TimeT, in at_time: TimeT, in execution_limit: unsigned long, in params: Any)`

Identical to the start operation except that the **at_time** parameter specifies an absolute time for the start of the first execution.

`pause()`

Pauses periodic execution.

`resume()`

Resumes periodic execution immediately.

`resume_at(in at_time: TimeT)`

Resumes periodic execution at a particular time.

`terminate()`

Terminates periodic execution.

`executions(): unsigned long`

Reports the number of periodic executions that have already been initiated.

Associations

No associations.

Constraints

No constraints.

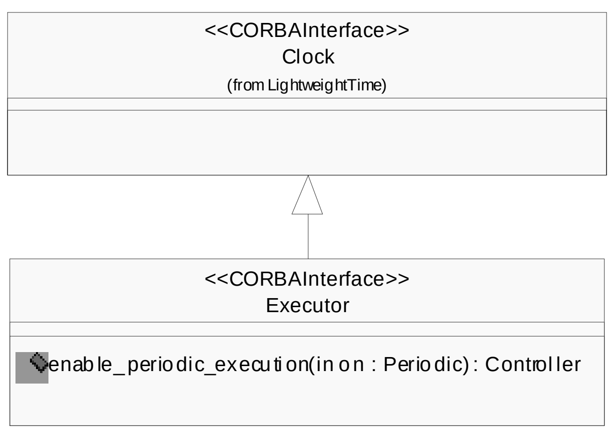
Semantics

This interface provides control over periodic execution. The appropriate object has been registered with the clock and must specialize the **Periodic** interface.

3.1.5.2 Executor

This interface is part of the optional minor conformance point “Support of Periodic Execution Control.”

Description



Register an object for periodic execution.

Attributes

No attributes.

Operations

`enable_periodic(in on: Periodic): Controller`

Registers an object that specializes the **Periodic** interface for periodic execution. The operation returns a reference to the associated **Controller** interface.

Associations

No associations.

Constraints

No constraints.

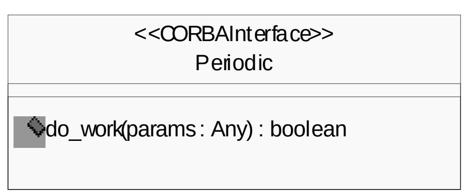
Semantics

The **Executor** is an interface for a factory that associates the specified object with a clock capable of supporting periodic execution. The registered object must specialize the **Periodic** interface. The **Executor** interface returns a reference to the **Controller** interface associated with this periodic execution.

3.1.5.3 *Periodic*

This interface is part of the optional minor conformance point “Support of Periodic Execution Control.”

Description



Make an object capable for periodic execution.

Attributes

No attributes.

Operations

`do_work(in params: Any): boolean`

The **do_work** operation will be periodically invoked by this service. Each invocation will be passed the type any value registered by the **start** or **start_at** operations on the Controller instance. The user implementation of the **do_work** operation should return a value of TRUE to continue periodic invocation; a value of FALSE will terminate periodic invocation.

Associations

No associations.

Constraints

No constraints.

Semantics

Instances of objects that are to be periodically executed must specialize and implement the **Periodic** interface. This means they must provide a **do_work** operation, and a means to enter a “ready to execute” state prior to registration with a clock.

3.2 Platform Specific Model: CORBA Service

3.2.1 Overview

The following sections specify a platform specific mapping of the Lightweight Time Service onto the CORBA platform. The resulting CORBA service is specified in CORBA IDL and represents a fully compatible subset of the Enhanced View of Time service, version 1.1

3.2.2 Minor Conformance Points

The platform specific model of the Lightweight Time Service supports the two minor conformance points of the platform independent model: *Support of Multiple Clocks* and *Support of Periodic Execution Control*. The selection of the corresponding features in the IDL definition is controlled by two preprocessor symbols controlling sets of conditional compilation preprocessor directives.

LW_TIME_HAS_SUPPORT_OF_MULTIPLE_CLOCKS

If this preprocessor symbol is defined, support for multiple clocks is activated by including the **ClockCatalog** interface and the **ClockEntry** structure.

LW_TIME_HAS_SUPPORT_OF_PERIODIC_EXECUTION_CONTROL

If this preprocessor symbol is defined, the **PeriodicExecution** module is enabled, which contains support for clock-controlled periodic execution.

3.2.3 LightweightTime Module

```
#include <TimeBase.idl>
#include <CosPropertyService.idl>
#pragma prefix "omg.org"
module LightweightTime
{
```

```
# ifndef _PRE_3_0_COMPILER_
   typeprefix "omg.org";
# endif // _PRE_3_0_COMPILER_

     interface Clock;
```

3.2.3.1 ClockProperty Module

```
module ClockProperty
{

     // the minimum set of properties to be supported for a clock
     typedef unsigned long Resolution;          // units = nanoseconds
     typedef short Precision;                   // ceiling of log_2(seconds
                                                // signified by least significant
                                                // bit of time readout)
     typedef unsigned short Width;              // no. of bits in readout -
                                                // usually <= 64
     typedef string Stability_Description;
     typedef short Coordination;
     const Coordination Uncoordinated = 0; // only static characterization
                                                 // is available

     typedef short TimeScale;
     // possible values for TimeScale ("pseudo-enumeration")
          const TimeScale Unknown          = -1;
          const TimeScale TAI              = 0;  // International Atomic Time
          const TimeScale UT0              = 1;  // diurnal day
          const TimeScale UT1              = 2;  // + polar wander
          const TimeScale UTC              = 3;  // TAI + leap seconds
          const TimeScale TT               = 4;  // terrestrial time
          const TimeScale TDB              = 5;  // Barycentric Dynamical Time
          const TimeScale TCG              = 6;  // Geocentric Coordinate Time
          const TimeScale TCB              = 7;  // Barycentric Coordinate Time
          const TimeScale Sidereal         = 8;  // hour angle of vernal equinox
          const TimeScale Local            = 9;  // UTC + time zone
          const TimeScale GPS              = 10; // Global Positioning System
          const TimeScale Other            = 0x7fff; // e.g. mission
     // end of pseudo-enumeration

     typedef string Comments;

}; // end of module ClockProperty

exception TimeUnavailable {};
```

3.2.3.2 Clock Interface

```
// the basic clock interface
interface Clock // a source of time readings
{
     readonly attribute CosPropertyService::PropertySet properties;
     readonly attribute TimeBase::TimeT current_time
```

```
        getRaises(TimeUnavailable);
    };
```

3.2.3.3 ClockCatalog Interface

```
#ifdef LWTIME_HAS_SUPPORT_OF_MULTIPLE_CLOCKS

    // alternative to Trader service (e.g., for embedded systems)
    //  Optional for system support of multiple clocks.
    interface ClockCatalog
    {
        struct ClockEntry
        {
            Clock       subject;
            string      name;
        };

        typedef sequence<ClockEntry> ClockEntries;
        exception UnknownEntry {};
        ClockEntry get_entry(in string with_name) raises (UnknownEntry);
        ClockEntries available_entries();
        void register(in ClockEntry entry);
        void delete_entry(in string with_name) raises (UnknownEntry);
    };

#endif  // LWTIME_HAS_SUPPORT_OF_MULTIPLE_CLOCKS
```

3.2.3.4 ControllableClock Interface

```
    // a controllable clock
    interface ControlledClock: Clock
    {
        exception NotSupported {};
        void set(in TimeBase::TimeT to)         raises (NotSupported);
        void set_rate(in float ratio)           raises (NotSupported);
        void pause()                            raises (NotSupported);
        void resume()                           raises (NotSupported);
```

```
        void terminate()                    raises (NotSupported);
    };
```

3.2.4 PeriodicExecution Module

```
    // Optional for Lightweight Time.

#ifdef LWTIME_HAS_SUPPORT_OF_PERIODIC_EXECUTION_CONTROL

    module PeriodicExecution
    {
```

3.2.4.1 Periodic Interface

```
        // (conceptually abstract) base for objects that can be
        // invoked periodically
        interface Periodic
        {
            boolean do_work(in any params); // return FALSE terminates
                                            // periodic execution
        };
```

3.2.4.2 Controller Interface

```
        // control object for periodic execution
        interface Controller
        {
            exception TimePast {};
            void start(in TimeBase::TimeT period,
                       in TimeBase::TimeT with_offset,
                       in unsigned long   execution_limit, // 0 = no limit
                       in any             params);
            void start_at(in TimeBase::TimeT period,
                       in TimeBase::TimeT at_time,
                       in unsigned long   execution_limit, // 0 = no limit
                       in any             params) raises (TimePast);
            void pause();
            void resume();
            void resume_at(in TimeBase::TimeT at_time) raises(TimePast);
            void terminate();
            unsigned long executions();
        };
```

3.2.4.3 Executor Interface

```
        // factory clock for periodic execution
        interface Executor : Clock
        {
```

```
            Controller enable_periodic_execution(in Periodic on);
        };

    }; // end of module PeriodicExecution

#endif  // LWTIME_HAS_SUPPORT_OF_PERIODIC_EXECUTION_CONTROL

}; //end of module LightweightTime

#endif // _LightweightTime_IDL_
```

Consolidated OMG IDL A

A.1 OMG IDL Listing

```
//File: CosClockService.idl
#ifndef _CosClockService_IDL_
#define _CosClockService_IDL_

// This module comprises the COS Clock service

#include <TimeBase.idl>
#include <CosPropertyService.idl>

#pragma prefix "omg.org"
module CosClockService
{

    interface Clock;

    module ClockProperty
    {   // the minimum set of properties to be supported for a clock
        typedef unsigned long Resolution; // units = nanoseconds
         typedef short Precision; // ceiling of log_2(seconds signified by least
                                          // significant bit of time readout)
        typedef unsigned short Width; // no. of bits in readout - usually <= 64
        typedef string Stability_Description;

        typedef short Coordination;
         const Coordination Uncoordinated = 0;    // only static characterization
                                                  // is available
        const Coordination Coordinated   = 1;     // measured against another
                                                  // source
         const Coordination Faulty       = 2;     // e.g., there is a bit stuck

        // the following are only applicable for coordinated clocks
        struct Offset
        {
```

```
        long long measured;   // units = 100 nanoseconds
        long long deliberate; // units = 100 nanoseconds
    };

     typedef short Measurement;
    const Measurement Not_Determined = 0;  // has not been measured
    const Measurement Discontinuous  = 1;    // e.g., one clock is paused
    const Measurement Available  = 2;            // has been measured

    typedef float Hz;
    struct Skew
    {
        Measurement available;
        Hz measured;   // only meaningful if available = Available - in Hz
        Hz deliberate; // in Hz
    };
    typedef float HzPerSec;
    struct Drift
    {
        Measurement available;
        HzPerSec measured;   // meaningful if available = Available
                                    // in Hz/sec
        HzPerSec deliberate; // in Hz/sec
    };

    typedef short TimeScale;
    const TimeScale Unknown      = -1;
    const TimeScale TAI          = 0;      // International Atomic Time
    const TimeScale UT0          = 1;      // diurnal day
    const TimeScale UT1          = 2;      // + polar wander
    const TimeScale UTC          = 3;      // TAI + leap seconds
    const TimeScale TT           = 4;      // terrestrial time
    const TimeScale TDB          = 5;      // Barycentric Dynamical Time
    const TimeScale TCG          = 6;      // Geocentric Coordinate Time
    const TimeScale TCB          = 7;      // Barycentric Coordinate Time
    const TimeScale Sidereal     = 8;      // hour angle of vernal equinox
    const TimeScale Local        = 9;      // UTC + time zone
    const TimeScale GPS          = 10;     // Global Positioning System
    const TimeScale Other        = 0x7fff;  // e.g. mission

    typedef short Stratum;
    const Stratum unspecified = 0;
    const Stratum primary_reference = 1;
    const Stratum secondary_reference_base = 2;

    typedef Clock CoordinationSource; // what clock is coordinating with
    typedef string Comments;
};

exception TimeUnavailable {};

// the basic clock interface
interface Clock // a source of time readings
{
    readonly attribute CosPropertyService::PropertySet properties;
```

```
    readonly attribute TimeBase::TimeT current_time
        getRaises(TimeUnavailable);
};

 enum TimeComparison
{
    TCEqualTo,
    TCLessThan,
    TCGreaterThan,
    TCIndeterminate
};

enum ComparisonType
{
    IntervalC,
    MidC
};

 enum OverlapType
{
    OTContainer,
    OTContained,
    OTOverlap,
    OTNoOverlap
};

valuetype TimeSpan;

// replaces UTO from CosTime
valuetype UTC
{
     factory init(in TimeBase::UtcT from);
    factory compose(in TimeBase::TimeT time,
                in unsigned long   inacclo,
                in unsigned short  inacchi,
                in TimeBase::TdfT  tdf);
    public TimeBase::TimeT time;
    public unsigned long   inacclo;
    public unsigned short  inacchi;
    public TimeBase::TdfT  tdf;

    TimeBase::InaccuracyT  inaccuracy();
    TimeBase::UtcT         utc_time();

    TimeComparison compare_time(in ComparisonType comparison_type,
                                      in UTC with_utc);
    TimeSpan interval();
};

// replaces TIO from CosTime
valuetype TimeSpan
{
    factory init   (in TimeBase::IntervalT from);
    factory compose(in TimeBase::TimeT lower_bound,
                    in TimeBase::TimeT upper_bound);
```

```
        public TimeBase::TimeT lower_bound;
        public TimeBase::TimeT upper_bound;
        TimeBase::IntervalT time_interval();
        OverlapType spans (
                in  UTC      time,
                out TimeSpan overlap
                );
        OverlapType overlaps (
                in TimeSpan  other,
                out TimeSpan overlap
                );
        UTC time ();
    };

    // replaces TimeService from CosTime
    interface UtcTimeService : Clock
    {
        UTC universal_time() raises(TimeUnavailable);
        UTC secure_universal_time() raises(TimeUnavailable);
        UTC absolute_time(in UTC with_offset) raises(TimeUnavailable);
};

    // alternative to Trader service (e.g., for embedded systems)
    interface ClockCatalog
    {
        struct ClockEntry
        {
            Clock          subject;
            string         name;
        };
        typedef sequence<ClockEntry> ClockEntries;
        exception UnknownEntry {};

        ClockEntry get_entry(in string with_name) raises (UnknownEntry);
        ClockEntries available_entries();
        void register(in ClockEntry entry);
        void delete_entry(in string with_name) raises (UnknownEntry);
    };

    // a controllable clock
    interface ControlledClock: Clock
    {
        exception NotSupported {};
        void set(in TimeBase::TimeT to)   raises (NotSupported);
        void set_rate(in float ratio)     raises (NotSupported);
        void pause()                      raises (NotSupported);
        void resume()                     raises (NotSupported);
        void terminate()                  raises (NotSupported);
    };

    // useful for building user synchronized clocks
    interface SynchronizeBase : Clock
    {
        struct SyncReading
```

```
    {
        TimeBase::TimeT local_send;
        TimeBase::TimeT local_receive;
        TimeBase::TimeT remote_reading;
    };

    SyncReading synchronize_poll(in Clock with_master);
};

interface SynchronizedClock;

exception UnableToSynchronize
{
    TimeBase::InaccuracyT minimum_error;
};

// allows definition of a new clock that uses the underlying hardware source
// of the existing clock but adjusts to synchronize with a master clock
interface Synchronizable : SynchronizeBase
{
    const TimeBase::TimeT Forever = 0xFFFFFFFFFFFFFFFF;

    SynchronizedClock new_slave
        (in Clock                          to_master,
        in TimeBase::InaccuracyT           to_within,
                                           // synchronization envelope
        in short                           retry_limit,
                                           // if unable to attain accuracy
        in TimeBase::TimeT                 minimum_delay_between_syncs,
                                           // limits network traffic,
                                           // Forever precludes auto resync
        in CosPropertyService::Properties properties
                                           // if null list, then inherit
                                           // properties of self
  ) raises (UnableToSynchronize);
};

// able to explicitly control synchronization
interface SynchronizedClock : Clock
{
    void resynch_now() raises (UnableToSynchronize);
};

module PeriodicExecution
{

    // (conceptually abstract) base for objects that can be invoked periodically
    interface Periodic
    {
        boolean do_work(in any params);
        // return FALSE terminates periodic execution
    };

    // control object for periodic execution
    interface Controller
```

```
            {
                exception TimePast {};
                void start
                    (in TimeBase::TimeT period,
                    in TimeBase::TimeT with_offset,
                    in unsigned long   execution_limit, // 0 = no limit
                    in any             params);
                void start_at
                    (in TimeBase::TimeT period,
                    in TimeBase::TimeT at_time,
                    in unsigned long   execution_limit, // 0 = no limit
                    in any             params) raises (TimePast);
                void pause();
                void resume();
                void resume_at(in TimeBase::TimeT at_time) raises(TimePast);
                void terminate();
                unsigned long executions();
            };

            // factory clock for periodic execution
            interface Executor : Clock
            {
                Controller enable_periodic_execution(in Periodic on);
            };
        };

    };
    #endif // _CosClockService_IDL_
```

Implementation Guidelines B

B.1 Introduction

This appendix contains advice to implementors. Appropriate documented handling of the criteria presented here is mandatory for conformance to the Basic Time Service conformance point.

B.2 Criteria to Be Followed for Secure Time

The following criteria must be followed in order to assure that the time returned by the **secure_universal_time** operation is in fact secure time. If these criteria are not satisfactorily addressed in an ORB, then it must return the TimeUnavailable exception upon invocation of the **secure_universal_time** operation of the **UtcTimeService** interface.

Administration of Time

Only administrators authorized by the system security policy may set the time and specify the source of time for time synchronization purposes.

Protection of Operations and Mandatory Audits

The following types of operations must be protected against unauthorized invocation. They must also be mandatorily audited:

- Operations that set or reset the current time.
- Operations that designate a time source as authoritative.
- Operations that modify the accuracy of the time service or the uncertainty interval of generated timestamps.

Synchronization of Time

Synchronization of time must be transmitted over the network. This presents an opportunity for unauthorized tampering with time, which must be adequately guarded against. Clock Service implementors must state how time values used for time synchronization are protected while they are in transit over the network.

Clock Service implementors must state whether or not their implementation is secure. Implementors of secure time services must state how their system is secured against threats documented in Chapter 15, Security Service Specification. They must also document how the issues mentioned in this section are addressed adequately.

Conformance Points C

There are two conformance points for this service.

C.1 Clock Service

This set of services provide for multiple clocks, access to UTC and mission clocks, and clock synchronization.

This conformance point requires the implementation of all types, valuetypes, and interfaces in the **CosClockService** module with the exception of the **PeriodicExecution** module.

C.2 Periodic Execution Service

This service is defined in Section 2.10, “PeriodExecution Service,” on page 2-18. The periodic execution service supports repeated execution of a method on an object.

This conformance point requires the implementation of the **CosClockService::PeriodicExecution** module.

Index