LOS ANGELES

COUNTYWIDE INFORMATION EXCHANGE NETWORK

RECOMMENDATIONS
FOR THE IMPLEMENTATION OF
NEW TRAFFIC CONTROL SYSTEM
COMMAND/DATA INTERFACE PROGRAMS

FINAL – REVISION 5

Prepared for:
Los Angeles County
Department of Public Works

Prepared by:

TRANSCORE

626 Wilshire Boulevard
Suite 818
Los Angeles, CA 90017

May 19th, 2006

TABLE OF CONTENTS

PAGE #

TABLE OF EXHIBITS

PAGE #

REVISION HISTORY

VERSION	DATE	DESCRIPTION
Final	5/24/2004	• Final (Original Submittal)
Final (Revision 1)	9/23/2004	• Response to BI Tran Questions • Prioritize the IEN's "Inbound" TCS CDI Data Requirements
Final (Revision 2)	11/19/2004	• Update IDL related sections with latest TransCore version
Final (Revision 3)	01/20/05	• Update Section 2.4 TCS CDI Event Format according to IEN implementation
Final (Revision 4)	01/03/06	• Series 2000 CDI Software Modications
Final (Revison 5)	5/19/06	• Minor changes to detector status field descriptions

1. INTRODUCTION

1.1 PURPOSE OF THE DOCUMENT

The Los Angeles County Information Exchange Network (IEN) uses command/data interface (CDI) programs as it's interconnect to various traffic control systems. A CDI allows the IEN to read a limited set of intersection, section, and detector data from a traffic control system (TCS), and enables the IEN to send a limited set of commands to intersections and sections on the TCS.

The IEN communicates with TCS CDI programs using the Common Object Request Broker Architecture (CORBA). In CORBA, *interfaces* are defined in *Interface Definition Language* (IDL) files. They define the functionality of a server program without specifying its implementation details. Interfaces expose (specify) methods and attributes that are implemented by CORBA servers. A TCS CDI must support the set of CORBA interfaces defined in this document.

This document will describe how CDI programs must implement these interfaces, and provide implementation guidelines for use by the developers of other TCS systems when they create their own TCS CDI applications. This document will use the TCS CDI that TransCore wrote for its own Series 2000 TCS as a working example.

1.2 INTENDED AUDIENCE

Readers of this document are assumed to be software developers who are implementing a new CDI for the IEN. They should have a working knowledge of CORBA programming in C++. They should also understand the basic concepts of a computerized traffic control system.

1.3 ASSUMPTIONS

The TransCore Site Server application is the IEN component that will communicate with TCS CDI programs. It uses the TAO ORB, which implements version 2.3 of the CORBA standard. TCS CDI Developers must use an ORB implementation that supports at least version 2.2 of the CORBA Specification.

The Los Angeles County DPW holds the copyright to the IEN source code developed specifically for this program. The County may grant developers of TCS CDI programs the right to view or copy the source code of the Series 2000 TCS CDI, which may be used as a guide to development of other CDIs.

2. TCS CDI INTERFACE REQUIREMENTS

The TCS CDI interfaces were designed to enable the IEN Site Server process to communicate with any traffic control system CDI. This could be a CDI for TransCore's Series 2000 TCS, a CDI for a BI Trans QuicNet system, the LA County TCS, or others.

While TCS implementations will vary, all TCS programs must meet the following requirements:

- TCS CDI programs must implement the objects defined in the CORBA interfaces that are defined in *Section 2.1: CORBA Interfaces.*
- TCS CDI programs must meet performance requirements imposed on them by how the IEN programs (the Site Server process in particular) use the interfaces. Performance requirements are covered in *Section 2.2: TCS CDI Performance Requirements*.
- TCS CDI programs must publish their *CommandAccessorFactory* and *DataAccessorFactory* objects to the CORBA naming service instance running on the Site Server that connects to the TCS CDI as described in *Section 2.3: Usage of the CORBA Naming Service.*
- TCS CDI programs must format data to be translated to the IEN as described in *Section 2.4: TCS CDI Event Format*.

This section of the document begins by describing the four (4) IDL files that define the IEN interface to TCS CDI programs. The section continues by describing how TransCore's Series 2000 TCS CDI implements the interface. CDI implementers may use the Series 2000 CDI as an example of a working system. So long as they properly implement the CORBA interfaces, they need not follow it in exact detail.

2.1 CORBA INTERFACES

The CDI must implement the CORBA interfaces in the IDL files described below. The interfaces define a contract between the IEN processes that use data from traffic control systems and TCS CDI programs.

The IDL files containing the interfaces that the TCS CDI must implement are:

- IENRTData.idl
- TCS.idl
- TCSData.idl
- TCSCommand.idl

Complete listings of the IDL files are provided in Appendix A.

2.1.1 IENRTData.idl

This file defines the IENRTData CORBA module, which contains basic type definitions and data structures for all TCS data exchanged by IEN programs. The *DeviceType* enumeration defines the kinds of traffic control system devices used in the IEN:

```
enum DeviceType
{
```

```
        DT_SYSTEM,
        DT_SCHEDULE,
        DT_INTERSECTION,
        DT_SECTION,
        DT_DETECTOR,
        DT_SIGN,
        DT_CAMERA,
        DT_HAR
};
```

Traffic control systems need only concern themselves with the following device types: DT_SYSTEM, DT_INTERSECTION, DT_SECTION, and DT_DETECTOR.

The file also defines the data structures that the CDI uses to send traffic control system data to the IEN. These are the *Event*, *EventSeq*, *EventGroup*, and *EventGroupSeq* structures.

2.1.1.1 Event

The *Event* structure represents one kind of data for one device that the TCS exports to the IEN. It contains an entity number, which represents the device ID, an event type, which defines the kind of data that the *Event* structure contains, and a timestamp denoting when the event was created. The rest of the structure contains areas that may contain long, short, octet, string, and double values. The event type value indicates to clients and servers the components of the *Event* structure that will be used for that particular event type.

See Section 2.3: TCS CDI Event Format for details of how the TCS CDI should create Event structures.

```
struct Event
{
    short            entityNumber;  // entity number (unique to system and event type)
    short            ienEventType;  // type of event (see above)
    long             timeStamp;     // in the format HHMMSS
    sequence<long>   longValues;    // sequence of 32-bit values
    sequence<short>  shortValues;   // sequence of 16-bit values
    sequence<octet>  octetValues;   // sequence of bytes
    string           stringValue;
    double           doubleValue;
};
```

2.1.1.2 EventSeq

The *EventSeq* sequence contains a set of Event objects. Each time that the Site Server requests data from the CDI process, it must return the requested data in an *EventSeq* sequence.

```
typedef sequence<Event> EventSeq;
```

2.1.1.3 EventGroup and EventGroupSeq

The IEN builds *EventSeq* structures that it receives from CDI programs into *EventGroup* and *EventGroupSeq* structure for distribution around the network. CDI programs do not have to build these structures.

2.1.2 TCS.idl

This file contains the definition of the TCS module. It includes the `IENRTData.idl` file. The `TCS.idl` file is included by both the `TCSData.idl` and `TCSCommand.idl` files.

2.1.2.1 *Mode* Enumeration

```
/// Modes of operation
    enum Mode
    {
      NORMAL,
      LOCAL_TOD,
      FREE,
      TOD,
      RESPONSIVE,
      MANUAL,
      RELEASE
    };
```

This enumeration defines the valid control modes that a CDI should report for an intersection controller, and the possible modes that the IEN may command for intersections and sections. See Table 2-1 below for an explanation of the enumeration values.

Table 2-1: Intersection Control Modes

CONTROL MODE	EXPLANATION
TOD	The central TCS chooses the timing pattern for the controller based on a central time-of-day schedule.
NORMAL	The central TCS chooses the timing pattern for the controller based on the timing pattern for the section and system that contain the intersection.
RESPONSIVE	The central TCS chooses the timing pattern for the intersection controller based on a traffic responsive algorithm.
FREE	The intersection controller should run free, with no programmed central timing pattern.
MANUAL	The intersection controller should run a timing pattern selected by a central TCS operator
LOCAL_TOD	The intersection controller selects its timing pattern based on its own time-of-day schedule.
RELEASE	The IEN uses the RELEASE control mode to tell a TCS that the IEN wishes to release external control of a device

2.1.2.2 SpecialFunction Enumeration

```
enum SpecialFunction

{

    FLASH,

    SPECIAL_FUNCTION_1,

    SPECIAL_FUNCTION_2,

    SPECIAL_FUNCTION_3,

    SPECIAL_FUNCTION_4

};
```

The *SpecialFunction* enumeration defines the special function commands that an operator may request for an intersection controller. It is not presently used, but is defined for future implementation of commands to activate or deactivate the special function outputs of a controller or put controllers into flash.

2.1.2.3 *Device* Structure

The *Device* structure defines a device identifier structure that contains a device type field and a device ID field.

```
/// Device identifier number

typedef short DeviceID;


/// Unique identifier for a device in a given TCS

struct Device

{

    IENRTData::DeviceType type;

    DeviceID   id;

};
```

2.1.2.4 DeviceList Sequence

The IEN uses the *DeviceList* sequence when requesting a list of available devices data from a TCS CDI, and when sending a command to a list of TCS devices.

```
/// List of Device elements

typedef sequence<Device> DeviceList;
```

2.1.2.5 *Version* Structure

The version structure contains software version information for a TCS CDI program. The *ConfigurationAccessor* interface contains two *Version* structures, one to identify the version of the IEN IDL expected by the CDI, and another to identify the version of the CDI software running on the TCS. The structure defines the major version, minor version, and revision level for the expected IDL and the version of the command or data accessor.

```
/// Version number (major.minor.revision)
struct Version
{
	short major;
	short minor;
	short revision;
};
```

2.1.2.6 *Status* Enumeration

The *Status* enumeration contains overall status information for a command or data interface to a traffic control system. The IEN expects that the CDI will report the TCS status as *SYSTEM_NORMAL* during normal operation of the TCS and the CDI. A return of any other status indicates that some or all TCS CDI features may not operate normally.

```
enum Status
{
	/// Running normally
	SYSTEM_NORMAL,

	/// Initializing; features may be unavailable or uninitialized
	SYSTEM_STARTING,

	/// Shutting down; features may be unavailable or no longer updated
	SYSTEM_STOPPING,

	/// TCS is not running
	SYSTEM_SHUTDOWN,

	/// TCS is unable to run
	SYSTEM_ERROR
};
```

2.1.2.7 ConfigurationAccessor Interface

This is a base interface for the *TCSCommandAccessor* interface, which the IEN uses to send commands to a TCS, and the *TCSDataAccessor* interfaces, which the IEN uses to get device data from a TCS. It defines version information, status information, and exported device information available for both interfaces.

Instructions for setting the values of these attributes are specified later in the document.

```
interface ConfigurationAccessor
{
	/// @return Version number for TCS CORBA interface
	readonly attribute Version interfaceVersion;

	/// @return Version of TCS software
	readonly attribute Version systemVersion;
```

```
        /// @return Name of TCS system
        readonly attribute string systemName;

        /// @return Current status of TCS
        readonly attribute Status systemStatus;

        /// @return list of configured devices of the given types
        DeviceList getAvailableDevices(in DeviceTypeList types);
};
```

2.1.3 TCSData.idl

The `TCSData.idl` file defines the TCSData module. It depends on the `TCS.idl` and `IENRtData.idl` files. It defines the *DataAccessor* interface, which the IEN Site Server program calls to get traffic control system data, and the *DataAccessorFactory* interface, which the IEN Site Server calls to get an instance of the *DataAccessor*.

2.1.3.1 *Code* Typedef

The TCS data interface file uses the *Code* typedef for types of device data requested by the IEN.

```
typedef short Code;
```

Section 2.4.1 describes the device data/event codes currently defined for the IEN.

2.1.3.2 *CodeList* Sequence

The *CodeList* sequence is used to pass a list of device data types into or out of the methods of the *DataAccessor* interface.

```
// List of data item code elements
typedef sequence<Code> CodeList;
```

2.1.3.3 DeviceDataTypes Structure

The *DeviceDataTypes* structure contains all of the data types that the IEN may request from the CDI for a particular device type. It allows the IEN to query the CDI to determine which data types the CDI supports.

```
    struct DeviceDataTypes
    {
        IENRTData::DeviceType type;
        CodeList dataTypes;
    };
```

2.1.3.4 DeviceDataTypeList Sequence

The *DeviceDataTypeList* sequence is returned from the CDI's deviceDataTypes method to indicate the complete set of device types that the CDI supports and the data types that the CDI can return to the IEN in a getDeviceEventDataList method call.

2.1.3.5 *DeviceCode* Structure

The *DeviceCode* structure identifies one device that the IEN is requesting from the TCS CDI, the kinds of data that the IEN is requesting for the device, and a flag indicating if change-only data is requested for the device.

```
struct DeviceCode
{
    TCS::Device device;
    CodeList    dataCodes;
    boolean     changedOnly;
};
```

2.1.3.6 DeviceCodeList Sequence

The IEN passes this sequence as a parameter to the getDeviceEventDataList method of the *DataAccessor* interface. It identifies all device and data types that the IEN is requesting from the TCS CDI at one time.

```
/// List of DeviceCode elements
typedef sequence<DeviceCode> DeviceCodeList;
```

2.1.3.7 DataAccessor Interface

The IEN programs call methods of the *DataAccessor* interface to get real-time traffic device data from TCS CDI programs.

```
/// Interface for retrieving data from TCS
interface DataAccessor: TCS::ConfigurationAccessor
{
    /// Instance name passed to DataAccessorFactory's
    /// CreateDataAccessor() method to create this instance.
    readonly attribute string clientName;

    /// Client calls this method when finished with this
    ///  DataAccessor.  Releases all resources associated with this
    /// instance.
    void destroy();

    /// @return the configured device list for this instance.
    TCS::DeviceList getDeviceList();

    /// Get the data type codes supported for all device types
    /// for which this CDI returns data.
    /// @return       supported data types for all supported devices
    DeviceDataTypeList deviceDataTypes ();

    /// @return Data items for input device list.
    ///
    /// @param devices  List of devices for which to get data.  Each
    ///                 entry in the list has a device ID, requested
    ///                 data types, and a changeOnly flag indicating
    ///                 if the method should retrieve only changed
    ///                 data (if true), or all known data (if false).
    /// @return         Sequence of IEN events containing requested
    ///                 requested data
    /// @throws         SystemStatusException if system not currently
    ///                 running
    /// @throws         Error if a device ID or data type in the
    ///                 device list is not supported.

    IENRTData::EventSeq getDeviceEventDataList(
```

```
                        in DeviceCodeList devices)
                        raises (TCS::SystemStatusException,
                                TCS::Error);
};
```

2.1.3.7.1 Configuration Retrieval Methods

The getDeviceList method returns the list of devices supported by the TCS CDI, including their identifiers and types. The deviceDataTypes method returns the data type codes returned by the CDI program for all supported device types. The clientName attribute returns the client name set for the DataAccessor when it was created by a DataAccessorFactory object, as discussed in *Section 2.1.3.8: DataAccessorFactory Interface* below.

2.1.3.7.2 Data Retrieval Method

The data retrieval method, *getDeviceEventDataList*, retrieves data for the requested list of devices. If a device's *changesOnly* parameter is true, the CDI should return only the requested data for the device that has changed since the CDI last checked it. If a device's *changesOnly* parameter is false, the CDI should retrieve all requested data for the device.

2.1.3.7.3 Destroy Method

The *destroy* method releases the resources owned by a given instance of the *DataAccessor* object. The IEN calls it when it is finished using the *DataAccessor* object.

2.1.3.8 DataAccessorFactory Interface

The IEN Site Server calls this interface when it wishes to begin getting intersection data from a traffic control system.

```
/// Interface for creating instances of DataAccessor
interface DataAccessorFactory
{
    /// Create an instance of DataAccessor.
    ///
    /// @param clientName  Text identifying the user
    ///                    of this interface.  For
    ///                    informational and diagnostic
    ///                    purposes only.
    ///
    /// @param option      Provides access to special
    ///                    functionality (generally for
    ///                    debugging or testing purposes).
    ///                    Always pass 0 unless you know
    ///                    what you're doing.
    /// @throws Error      If the client name is empty or the
    ///                    option is not supported

    DataAccessor createDataAccessor(    in string clientName,
                                        in long   option )
                                        raises ( TCS::Error );
};
```

2.1.4 TCSCommand.idl

The `TCSCommand.idl` file defines the TCSCommand module. It depends on the `TCS.`idl and `IENRtData.idl` files. It defines the *CommandAccessor* interface, which the IEN Site Server program calls to send commands from the IEN to traffic control devices on a TCS, and the *CommandAccessorFactory* interface, which the IEN Site Server calls to get an instance of the *CommandAccessor*.

2.1.4.1 CommandAccessor Interface

The IEN sends commands to traffic control system command/data interface programs using the *CommandAccessor* interface.

```
/// Interface that allows clients to send commands to Series 2000
interface CommandAccessor: TCS::ConfigurationAccessor
    {
        /// Text passed to DataAccessorFactory::CreateDataAccessor()
        /// to create this instance.
        readonly attribute string clientName;

        /// Client calls this method when finished with this CommandAcceptor.
        /// Releases all resources associated with this instance.
        void destroy();

       /// Change CDI plan number
        ///
        /// @param devices    List of devices for which to change the plan
        /// @param planNumber New plan number for the requested devices
        void setCDIPlan(
            in TCS::DeviceList devices,
            in short           planNumber
        ) raises (
            CommandsNotAccepted,
            InvalidPlanNumber,
            TCS::UnknownDevices,
            TCS::SystemStatusException,
            TCS::Error
        );

        /// Change operational mode of specified devices.
        ///
        /// @param devices    List of devices for which to change the mode
        /// @param newMode    New operational mode for the devices on the list
        void changeMode(
            in TCS::DeviceList devices,
            in TCS::Mode       newMode
        ) raises (
            CommandsNotAccepted,
            InvalidMode,
            TCS::UnknownDevices,
            TCS::SystemStatusException,
            TCS::Error
        );

        /// Release IEN control of a list of TCS devices
        ///
        /// @param devices    List of devices for which to release IEN control
        void releaseControl(
            in TCS::DeviceList devices
        ) raises (
            TCS::UnknownDevices,
```

```
            TCS::SystemStatusException,
            TCS::Error
        );
};
```

2.1.4.1.1 setCDIPlan Method

The *setCDIPlan* method changes the timing plan or timing pattern number being run by a list of intersection controllers and sections on the TCS. The CDI should make a "best effort" attempt to command the controllers and sections to run the requested timing pattern or plan number. If it cannot implement the requested command on one of the requested devices, it should raise one of the exceptions defined in the module.

2.1.4.1.2 changeMode Method

changeMode changes the operating mode of the devices on the list to the requested mode. See Table 2-1 for a list of possible operating modes for a section or intersection. If the CDI does not support a requested mode at all, it should raise the *InvalidMode* exception.

2.1.4.1.3 releaseControl Method

This method ends external control of all devices on the requested list. The CDI should return the devices to the state desired by the TCS when the CDI receives the *releaseControl* method call.

2.1.4.2 CommandAccessorFactory Interface

The IEN Site Server calls this interface when it wishes to begin sending commands to devices on a traffic control system.

```
/// Interface for creating instances of CommandAcceptor
interface CommandAccessorFactory
{
       /// Create an instance of DataAccessor.
       ///
       /// @param clientName  Text identifying the user
       ///                    of this interface.  For
       ///                    informational and diagnostic
       ///                    purposes.
       ///
       /// @param option      Provides access to special
       ///                    functionality (generally for
       ///                    debugging or testing purposes).
       ///                    Always pass 0 unless you know
       ///                    what you're doing.
       CommandAccessor createCommandAccessor(
              in string clientName,
              in long   option
       ) raises (TCS::Error);
```

2.2 TCS CDI PERFORMANCE REQUIREMENTS

The TCS CDI has two sets of performance requirements. One set of requirements pertains to handling data requests from the site server, and another to handling command requests from the IEN.

2.2.1 Data Access Requirements

The TCS CDI must be able to completely process a call to the *DataAccessor* interface's *getDeviceEventDataList* method within a half second.

order:

- system
- list
- CDI

The IEN requests configuration data on a different fraction ($1/60^{th}$) of the TCS devices each second. Consequently, it requests configuration data on all devices exported by the TCS once per minute.

2.2.2 Data Reporting Requirements

When the Site Server queries a CDI for configured devices, it must return a list containing one or more *Event* structures with data for the requested devices, as described in Section 2.4. Not all traffic control systems can respond with the complete complement of data requested by the Site Server. For examples, some systems poll controllers less frequently than once per second, while others do not poll controllers unless specifically requested to do so by an operator. This section describes which events the CDI must return when the Site Server requests them, and which the CDI may omit.

Event types requested by the IEN fall in the following prioritized categories:

- **Required** – the CDI must always return events into this category when the Site Server requests them.
- **Required** – (if a certain feature is supported) The CDI must return events in this category when the Site Server requests them if the CDI supports the related feature (system detectors and/or sections).
- **Highly Desirable** – The CDI should return events in this category to the Site Server if possible, as they provide much of the utility of the IEN to users.
- **Desirable** – Desirable items returned by the CDI are helpful but not critical to the use of the IEN.

Please note that it is highly desirable that the IEN receive data from the TCS CDI on a once-per-second basis. However, the IEN will be capable of receiving data at whatever frequency the TCS supports.

2.2.2.1 IEN_INTERSECTIONINFO

The CDI must always return a full IEN_INTERSECTIONINFO structure for an intersection controller when the Site Server requests it.

2.2.2.2 IEN_INTERSECTIONRTSTATUS

The CDI must always return an IEN_INTERSECTIONRTSTATUS structure for an intersection controller when the Site Server requests it. The CDI may set fields other than the cycle counter and reference cycle counter to –1 if values are unknown.

2.2.2.3 IEN_INTERSECTIONRTSUMMARY

The CDI must always return this event when the Site Server requests it. The CDI must return its best guess of the timing plan ID, controller alarms, controller response state, controller communication state, and whether main street green is active or not at the time of the request. If either the communication status values indicate a communication error to the controller, the IEN will ignore the rest of the controller data until both fields indicate the CDI is communicating with the controller.

It is highly desirable for the CDI to return usable data for the signal control mode, cycle length, and desired and actual offset fields. The preemption type field is desirable, but may be set to IPT_NO_PREEMPT if the controller's preemption type is not known by the TCS at the time of the request.

2.2.2.4 IEN_PHASE_STATEDATA

It is highly desirable for the CDI to return this event type when the Site Server requests it. If the CDI returns this data, it should send an event to the IEN every time that it is requested. If no phases have changed state since the previous request, the CDI should send the same data to the Site Server as before.

This data is used to drive IEN displays that show which phases are active on a controller (controller headers and phase arrows in intersection diagrams and phase times in intersection detail displays).

2.2.2.5 IEN_PEDPHASE_STATEDATA

It is desirable for the CDI to return this event when the Site Server requests it. If the CDI returns this data, it should send an event to the IEN every time that it is requested. If no pedestrian displays have changed state since the previous request, the CDI should send the same data as before.

The data in this event drives pedestrian walk/don't walk symbols and pedestrian phase displays in intersection header controls in intersection diagrams and the pedestrian phase displays in intersection detail displays.

2.2.2.6 IEN_VEHCALL_STATEDATA

It is desirable for the CDI to return this event when the Site Server requests it. If the CDI returns this data, it should send an event to the IEN every time it is requested. If no vehicle calls have changed state since the previous request, the CDI should send the same data as before.

The data in this event drives actuation detector symbols and the vehicle call displays in the intersection header control in intersection diagrams, and the vehicle call displays in intersection detail displays.

2.2.2.7 IEN_LASTCYCLE_PHASEDATA

It is highly desirable that the CDI should return this event when requested by the Site Server. The CDI should return this data at least once per cycle, preferably at the beginning of the controller's new cycle.

The data in this event drives the displays of the phase times in the last cycle shown in the intersection header controls in intersection diagrams and in intersection detail displays.

2.2.2.8 IEN_TP_PHASEDATA

It is highly desirable that the CDI should return this event when requested by the Site Server. The CDI should update this data when the controller changes the maximum green times allotted to any of its phases, such as when it changes timing plans or phase timing parameters. The data changes relatively rarely, generally when a user changes the controller's global phase timing parameters, although for some controllers phase maximum can change with timing plans.

The data in this event drives the displays of maximum phase times in the intersection header controls in intersection diagrams and in intersection detail displays.

2.2.2.9 IEN_DETECTORINFO

If the CDI supports any system detectors, it must return this event when the Site Server requests it. For systems that do not support a volume plus weighted occupancy calculation, 30 is a reasonable default value for the weighting factor.

2.2.2.10 IEN_DETECTORSTATE

If the CDI supports any system detectors, it must return this event when the Site Server requests it. The CDI should return new information in this event each time after the TCS uploads data from a system detector.

2.2.2.11 IEN_SECTIONINFO

If the CDI supports sections, it must return this event when the Site Server requests it.

2.2.2.12 IEN_SECTIONSTATE

If the CDI supports sections, it must return this event when the Site Server requests it.

2.2.3 Command Execution Requirements

The TCS CDI must be able to process the following method calls to its *CommandAccessor* interface within 10 seconds:

- setCDIPlan
- changeMode
- releaseControl

2.3 USAGE OF THE CORBA NAMING SERVICE

The TCS CDI must publish references to its objects in the connecting IEN Site Server's *CORBA Naming Service*. The IEN Site Server program uses the naming service to locate the TCS CDI CORBA objects. In the event of a naming service failure, the CDI must periodically attempt to rebind its objects to the naming service. The attempt interval should be 5 minutes or less.

2.3.1 IEN Naming Service Location

A TCS CDI implementation must publish its *CommandAccessorFactory* and *DataAccessorFactory* objects to the naming service instance running on port 14444 on the Site Server machine at the site where the TCS CDI is running. The IEN network design requires that there be a Site Server at every site that runs a TCS CDI.

The CDI should set its name service reference to the equivalent of the following URI:

```
corbaloc:iiop:<site server name>:14444/NameService
```

The <site server name> string should be replaced by the network node name of the Site Server at the IEN site where the TCS CDI runs.

2.3.2 Published Names

The TCS CDI must publish two (2) names to the naming service, one for its *CommandAccessorFactory* object, and the second for its *DataAccessorFactory* object. Each name should have one element, with the ID and KIND fields set to the strings shown in Table 2-2.

Table 2-2: CDI Names in the Naming Service

OBJECT	ID	KIND
CommandAccessorFactory	TCSCDICmd<site ID>	Site<site ID>
DataAccessorFactory	TCSCDIData<site ID>	Site<site ID>

The <site ID> string should be replaced with the ID number of the site at which the TCS CDI is running.

2.4 TCS CDI EVENT FORMAT[1]

The TCS CDI must provide traffic control system event data to the Site Server program from its *DataAccessor* interface in *Event* structures. These structures are defined in the IENRTData module of the IENRTData.idl file. The IENRTData.IDL file is discussed within *Section 2.1.1: IENRTData.idl* above and a listing is provided in Appendix A. The Site Server program then relays this event data to user workstations at the local site and to the corridor server for distribution to other sites.

Event structures have the format shown in Table 2-3. All enumeration types used in these events are defined in the file sgv/src/common/tcscdi/src/IENDataDefine.hpp.

Table 2-3: Event Format

FIELD	FORMAT	CONTENT
EntityNumber	2-byte signed integer (short)	Unique ID of the device to which the event data applies
IenEventType	2-byte signed integer (short)	Type of data that the event contains
Timestamp	4-byte signed integer (long)	Event timestamp, in the format HHMMSS (1:35:15 PM would be represented as 133515)
LongValues	CORBA sequence (expandable array) of 4-byte signed integers	Depends on the event type
ShortValues	CORBA sequence of 2-byte signed integers	Depends on the event type
OctetValues	CORBA sequence of 1-byte signed integers	Depends on the event type
StringValue	CORBA string	Depends on the event type
DoubleValue	CORBA double floating point value	Depends on the event type

2.4.1 Event Types

The file sgv/src/common/network/src/ientypedef.hpp contains the set of event types defined for the IEN, as shown in Table 2-4. The event types are used in variables given the *Code* typedef in the IDL files.

Table 2-4: IEN Event Types

EVENT TYPE	ENTITY	PRIORITY
IEN_COMMANDRETURN	Command	N/A
IEN_INTERSECTIONINFO	Intersection	Required
IEN_INTERSECTIONRTSTATUS	Intersection	Required
IEN_INTERSECTIONRTSUMMARY	Intersection	Required Highly Desirable
IEN_PHASE_STATEDATA	Intersection	Highly Desirable

[1] Unless otherwise specified, all tables listed in the sub-sections of 2.4 are used for descriptive purposes; actual enumeration values are defined in the relevant header files.

EVENT TYPE	ENTITY	PRIORITY
IEN_PEDPHASE_STATEDATA	Intersection	Desirable
IEN_VEHCALL_STATEDATA	Intersection	Desirable
IEN_LASTCYCLE_PHASEDATA	Intersection	Highly Desirable
IEN_TP_PHASEDATA	Intersection	Highly Desirable
IEN_DETECTORINFO	Detector	Required (If Supported)
IEN_DETECTORSTATE	Detector	Required (If Supported)
IEN_SECTIONINFO	Section	Required (If Supported)
IEN_SECTIONSTATE	Section	Required (If Supported)

2.4.2 Command Events

2.4.2.1 IEN_COMMANDRETURN Event Type

This event type is not used.

2.4.3 Intersection Events

The following subsections show the format of event data for all intersection commands. They show only the content of the fields of the Event structure that change. The header fields, entityNumber and timeStamp, which do not change their value for the different types of events, are omitted.

2.4.3.1 IEN_INTERSECTIONINFO Event Type

The TCS CDI should return the IEN_INTERSECTIONINFO event (as described in Table 2-5) to the IEN Site Server when a new intersection controller is added to the TCS or any of the configuration data for an intersection controller changes. The CDI should return this event with the Intersection ID number of "-1" (please refer to Table 2-5 Field shortValues[0]) for an intersection controller that has been removed from the TCS or does not exist in the TCS.

Table 2-5: Intersection Configuration Information

FIELD	CONTENTS
eventType	IEN_INTERSECTIONINFO
longValues	None
shortValues[0]	Intersection ID number, -1 if no intersection with the ID exists
shortValues[1]	ID number of section containing this intersection, -1 if none
shortValues[2]	Seconds between poll attempts to the intersection controller
OctetValues[0]	Low byte of controller type string, as defined in Table 2-6
OctetValues[1]	Next byte of controller type string
…	
OctetValues[n]	Last byte of controller type string, where n+1 is the length of the string. This should not be a 0 byte

FIELD	CONTENTS
StringValue	Description of the intersection controller. The description string should be of the form "Main Street @ Cross Street"
doubleValue	None

Currently defined controller type strings are listed in Table 2-6.

Table 2-6: IEN Controller Type Strings

CONTROLLER TYPE STRING
AB3418 Protocol
BI Tran
BI Tran 207LRT
BI Tran 222WP
BI Tran CIC
Central Plan Implementor
NTCIP Protocol
Series 2000 RCU
Series 2000 RCU LRT
Traconex TMP-390
Traconex TMP-390CJ
Traconex TMP-390M
Wapiti W4HC11
Wapiti W4IKS
Wapiti W4LRT
Wapiti W4LRT+
Wapiti W9FT

2.4.3.2 IEN_INTERSECTIONRTSTATUS Event Format

The CDI should return the IEN_INTERSECTIONRTSTATUS event (as described in

Table 2-7) to the IEN Site Server for an intersection each second in which the Site Server requests data for the intersection controller.

Table 2-7: Intersection Real-Time Status Data

FIELD	CONTENTS
eventType	IEN_INTERSECTIONRTSTATUS
longValues[0]	# of communication attempts to the intersection controller (-1 if unknown)
longValues[1]	# of good responses received from the intersection controller (-1 if unknown)
longValues[2]	# of bad responses received from the intersection controller (-1 if unknown)
longValues[3]	# of timeouts waiting for responses from the intersection controller (-1 if unknown)
shortValues[0]	Cycle counter, seconds since start of cycle
shortValues[1]	Time in seconds since reset of attempts counter (-1 if unknown or N/A)
shortValues[2]	Time in seconds since reset of good responses counter (-1 if unknown or N/A)
shortValues[3]	Time in seconds since reset of bad responses counter (-1 if unknown or N/A)
shortValues[4]	Time in seconds since reset of timeout counter (-1 if unknown or N/A)
shortValues[5]	Reference cycle counter for the intersection (cycle counter for controller with same cycle length but offset 0)
octetValues	None
stringValue	None
doubleValue	None

2.4.3.3 IEN_INTERSECTIONRTSUMMARY Event Format

The CDI should return the IEN_INTERSECTIONRTSUMMARY event (as described in Table 2-8) to the IEN Site Server whenever any of the data listed in the table changes for a given intersection controller.

Table 2-8: Intersection Real-Time Summary Data

FIELD	CONTENTS
EventType	IEN_INTERSECTIONRTSUMMARY
longValues[0]	Signal control mode (see Table 2-9)
longValues[1]	Intersection signal state (see Table 2-10)
longValues[2]	Controller response state: ICR_RESPONDING if controller responding to communication, ICR_NOT_RESPONDING if not
longValues[3]	Preemption type, defined in Table 2-11
longValues[4]	Controller alarms, a set of 0 or more bit masks fromTable 2-12. Multiple bit flags may be bitwise OR'ed together.
longValues[5]	Main street green active. 1 if the main street green phase is presently active, 0 if not, -1 if unknown.
longValues[6]	Communication state for the intersection controller, as shown in Table 2-13
longValues[7]	Timing plan ID number
longValues[8]	Desired cycle length, in seconds

FIELD	CONTENTS
longValues[9]	Desired offset, in seconds
longValues[10]	Actual offset, in seconds
ShortValues	None
OctetValues	None
StringValue	None
DoubleValue	None

Legal control modes for intersection controllers and sections follow in Table 2-9. They are drawn from the file sgv/src/common/tcscdi/src/IENDataDefine.hpp..

Table 2-9: Intersection Control Modes

CONTROL MODE	EXPLANATION
ISC_OTHER_NO_ADDITIONAL	Control mode other than the defined values
ISC_OTHER_ADDITIONAL	Unknown coordination mode, more information available
ISC_FREE	Free timing
ISC_FIXED_TIME	Controller is running fixed-time plan
ISC_TIME_BASE_COORDINATION	Controller is running a coordinated plan
ISC_ACTUATED	Controller is fully actuated
ISC_SEMI_ACTUATED	Controller is semi-actuated
ISC_CRITICAL_INTERSECTION_CONTROL	Controller is adjusting split times in its plan based on detector data
ISC_TRAFFIC_RESPONSIVE	Controller timing plan selected based on detector data (traffic responsive)
ISC_ADAPTIVE	Controller timing plan, cycle length, offset, and split times based on detector data (adaptive)
ISC_TRANSITION	Controller is in transition between timing plans
ISC_EXTERNAL	Intersection timing plan requested by external user

The valid signal states for intersection controllers are described in Table 2-10. The values are defined in the file sgv/src/common/tcscdi/src/IENDataDefine.hpp.

Table 2-10: Intersection Signal States

SIGNAL STATE	EXPLANATION
ISS_OTHER_NO_ADDITIONAL	Signal state other than the defined values
ISS_OTHER_ADDITIONAL	Signal state other than the defined values, more information available
ISS_NORMAL_OPERATION	Signal in normal operation state
ISS_FLASH	Signal in flashing operation state
ISS_PREEMPTION	Signal in preemption operation state

SIGNAL STATE	EXPLANATION
ISS_CONFLICT_FLASH	Signal in conflict flashing state

Legal preemption types for intersection controllers follow in Table 2-11, also drawn from the file sgv/src/common/tcscdi/src/IENDataDefine.hpp.

Table 2-11: Intersection Preemption Types

PREEMPTION TYPE	EXPLANATION
IPT_OTHER_NO_ADDITIONAL	Preemption type other than the defined values
IPT_OTHER_ADDITIONAL	Preemption type other than the defined values, more information available
IPT_NO_PREEMPT	No preemption presently active
IPT_GENERAL_PREEMPT	General preemption active
IPT_BRIDGE_PREEMPT	Bridge preemption active
IPT_EV_PREEMPT	Emergency vehicle preemption active
IPT_LRT_PREEMPT	Light rail transit preemption active
IPT_RR_PREEMPT	Railroad preemption active

Table 2-12 defines the alarm bit flags defined for intersection controllers.

Table 2-12: Intersection Controller Alarm Bit Flags

CONTROLLER ALARM BIT FLAG	EXPLANATION
ICA_NO_ALARM	No alarms currently defined
ICA_CONFLICT_FLASH_ALARM	The controller is in flash because of a conflict
ICA_CABINET_DOOR_OPEN_ALARM	The controller's cabinet door is open
ICA_TRANSITION_ALARM	The controller is in transition
ICA_INTERNAL_ERROR_ALARM	The controller has detected an internal error
ICA_FLASH_ALARM	The controller is in flash for a reason other than a conflict

Table 2-13 defines the possible communication state values for an intersection controller.

Table 2-13: Intersection Controller Communication States

CONTROLLER COMMUNICATION STATE	EXPLANATION
ICS_COMM_UNKNOWN	State of communication to the controller cannot be presently determined
ICS_COMM_OTHER	Unspecified other communication state
ICS_COMM_GOOD	The TCS is presently communicating properly with the controller
ICS_COMM_BAD	The TCS has determined there is a problem communicating with the controller

2.4.3.4 IEN_PHASE_STATEDATA Event Type

The CDI should return the IEN_PHASE_STATEDATA event (as described in Table 2-14) to the IEN Site Server when the state of any of the defined phases for an intersection controller changes. This event should contain the IDs of phases that are currently green. It should not contain the IDs of phases that are red, yellow, or inactive in the current timing pattern.

Table 2-14: Intersection Controller Phase State Data

FIELD	CONTENTS
EventType	IEN_PHASE_STATEDATA
LongValues	None
ShortValues	None
octetValues[0]	Active phase ID 1 (If there is no phase in active state, the length of octetValues sequence should be set to 1 and the value in this field set to "0")
octetValues[1]	Active phase ID 2
octetValues[2]	Active phase ID 3
…	
octetValues[n - 1]	Active phase ID n
StringValue	None
DoubleValue	None

2.4.3.5 IEN_PEDPHASE_STATEDATA Event Type

The CDI should return the IEN_PEDPHASE_STATEDATA event (as described in Table 2-15) to the IEN Site Server when the walk signal state changes for any defined phase in the intersection controller. This event should contain the IDs of phases that are currently displaying walk signals. It should not contain the IDs of phases that are not showing walk signals or that are inactive.

Table 2-15: Intersection Pedestrian Signal States

FIELD	CONTENTS
eventType	IEN_PEDPHASE_STATEDATA
longValues	None
shortValues	None
octetValues[0]	Active pedestrian phase ID 1 (If there is no pedestrian phase in active state, the length of octetValues sequence should be set to 1 and the value in this field set to "0")
octetValues[1]	Active pedestrian phase ID 2
octetValues[2]	Active pedestrian phase ID 3
…	
octetValues[n - 1]	Active pedestrian phase ID n
stringValue	None
doubleValue	None

2.4.3.6 IEN_VEHCALL_STATEDATA Event Type

The CDI should return the IEN_VEHCALL_STATEDATA event (as described in Table 2-16) to the IEN Site Server when any of the actuation detectors associated with a controller changes its actuation state. This event should contain only the IDs of phases that have active actuation detectors.

Table 2-16: Intersection Actuation Detector States

FIELD	CONTENTS
eventType	IEN_VEHCALL_STATEDATA
longValues	None
shortValues	None
octetValues[0]	Active phase ID 1 (If there is no phase in active state, the length of octetValues sequence should be set to 1 and the value in this field set to "0")
octetValues[1]	Active phase ID 2
octetValues[2]	Active phase ID 3
…	
octetValues[n - 1]	Active phase ID n
stringValue	None
doubleValue	None

2.4.3.7 IEN_LASTCYCLE_PHASEDATA Event Type

The CDI should return the IEN_LASTCYCLE_PHASEDATA event (as described in Table 2-17) to the IEN Site Server at the end of an intersection controller's cycle. It should contain the total green time for all phases that were active in the controller during the just-completed cycle.

Table 2-17: Phase Data for Previous Intersection Cycle

FIELD	CONTENTS
eventType	IEN_LASTCYCLE_PHASEDATA
longValues[0]	Total phase time for all phases in last cycle
longValues[1]	Phase ID 1
longValues[2]	Green time in seconds for phase ID 1 in the controller's last cycle
longValues[3]	Phase ID 2
LongValues[4]	Green time in seconds for phase ID 2 in the controller's last cycle
…	
longValues[2n-1]	Phase ID n (where n is the highest ID of any phase active in the controller in the last cycle)
longValues[2n]	Green time in seconds for phase ID n in the controller's last cycle
shortValues	None
octetValues	None

FIELD	CONTENTS
stringValue	None
doubleValue	None

2.4.3.8 IEN_TP_PHASEDATA Event Type

The CDI should return the IEN_TP_PHASEDATA event (as described in Table 2-18) to the IEN Site Server when the maximum permissible green time changes for any of the phases defined in the controller.

Table 2-18: Controller Phase Maximum Data

FIELD	CONTENTS
EventType	IEN_TP_PHASEDATA
LongValues	None
shortValues	None
OctetValues[0]	Phase ID 1
OctetValues[1]	Maximum time in seconds for phase ID 1
…	
OctetValues[2n - 2]	Phase ID n (where n is the largest ID of a phase defined in the controller's current timing parameters)
OctetValues[2n - 1]	Maximum time in seconds for phase n
StringValue	None
doubleValue	None

2.4.4 Detector Events

2.4.4.1 IEN_DETECTORINFO Event Type

CDI should return the IEN_DETECTORINFO event (as described inTable 2-19) to the IEN Site Server for a detector if a detector is created or any of the detector's configuration data changes. It should send the value of "-1" in field shortValues[0] if the requested detector is deleted from the TCS or does not exist in the TCS.

Table 2-19: Detector Configuration Information

FIELD	CONTENTS
EventType	IEN_DETECTORINFO
longValues[0]	Detector data averaging period, in seconds
ShortValues[0]	Detector ID, should set to the same value as Event.entityNumber when the requested detector exists, or "-1" when the detector does not exist in the TCS
octetValues[0]	Detector class. Should always be DC_SYSTEM, as shown in Table 2-20.
octetValues[1]	Detector type, from Table 2-21
octetValues[2]	Direction of traffic flow over the detector, as shown in Table 2-22

FIELD	CONTENTS
octetValues[3]	Lane number for traffic passing over the detector. The innermost lane on the roadway is lane 1, the next lane to the right is lane 2, etc.
stringValue	Name of the roadway that contains the detector
doubleValue	Weighting factor (K) for volume + weighted occupancy calculations

Detector classes are defined in the file sgv/src/common/idl/IENTCSData.idl. They are shown here in Table 2-20.

Table 2-20: IEN Detector Classes

DETECTOR CLASS	EXPLANATION
DC_OTHER_NO_ADDITIONAL	Other class with no additional information
DC_OTHER_ADDITIONAL	Other class with additional information (not presently supported)
DC_STOP_BAR	Intersection stop bar detector
DC_SYSTEM	Arterial traffic detector
DC_PEDESTRIAN	Arterial pedestrian detector
DC_ADAPTIVE	Arterial traffic detector used for adaptive control
DC_CALL	Arterial vehicle call (actuation) detector
DC_EXTENSION	Arterial extension detector
DC_MAINLINE	Mainline freeway detector
DC_REVERSIBLE_LANE	Reversible lane detector
DC_RAMP_DEMAND	Freeway ramp demand detector
DC_RAMP_MERGE	Freeway ramp merge detector
DC_RAMP_PASSAGE	Freeway ramp passage detector
DC_RAMP_QUEUE	Freeway ramp queue detector

Detector types (as described in Table 2-20) mean the mechanism that the detector uses to detect vehicles. They are defined in the file sgv/src/common/idl/IENTCSData.idl.

Table 2-21: IEN Detector Types

DETECTOR TYPE	EXPLANATION
DT_OTHER_NO_ADDITIONAL	Other detection mechanism, no additional information on what
DT_OTHER_ADDITIONAL	Other detection mechanism with additional explanatory information (not presently supported)
DT_INDUCTIVE_LOOP	Inductive loops
DT_MAGNETIC	Magnetic detection
DT_MAGNETOMETERS	Detection by magnetometers
DT_PRESSURE_CELLS	Detection by pressure cells
DT_MICROWAVE_RADAR	Detection by microwave radar
DT_ULTRASONIC	Detection using ultrasound
DT_VIDEO_IMAGE	Detection using video images

DETECTOR TYPE	EXPLANATION
DT_LASER	Detection using laser ranging
DT_INFRARED	Detection using infrared light
DT_ROAD_TUBE	Detection using a road tube

The detector direction codes should carry the values and meanings described in Table 2-22..

Table 2-22: IEN Detector Direction Codes

CODE VALUE	CODE MEANING
0	Eastbound
1	Westbound
2	Southbound
3	Northbound
4	Southeast bound
5	Southwest bound
6	Northeast bound
7	Northwest bound
8	Outbound
9	Inbound
10	None

2.4.4.2 IEN_DETECTORSTATE Event Type

The IEN should return the IEN_DETECTORSTATE event (as described in Table 2-23) to the IEN Site Server for each detector immediately after new data has been uploaded from the field into the TCS for that detector.

Table 2-23: Detector Status Information

FIELD	CONTENTS
eventType	IEN_DETECTORSTATE
longValues	None
longValues[0]	Volume from yhr most recent upload, in units of vehicles per hour
longValues[1]	Average volume, in units of vehicles per hour. Averaging period given in IEN_DETECTORINFO event.
longValues[2]	Volume in vehicles per hour + weighted occupancy, for volume and occupancy from the most recent upload. The weighting factor used is the weighting factor reported in the IEN_DETECTORINFO event.
longValues[3]	Average volume in vehicles per hour + weighted occupancy, for volume and occupancy in the averaging period. The averaging period and weighting factor used are reported in the IEN_DETECTORINFO event.

FIELD	CONTENTS
shortValues[0]	Detector status, from Table 2-24
shortValues[1]	Speed data from the most recent upload, in miles per hour
shortValues[2]	Average speed data over the averaging period, in miles per hour. Averaging period comes from the IEN_DETECTORINFO event.
shortValues[3]	Occupancy data from the most recent upload, in percent (0-100)
shortValues[4]	Average occupancy data for the detector, in percent (0-100). The averaging period is reported in the IEN_DETECTORINFO event.
stringValue	None
doubleValue	None

The detector status codes (as described in) are defined in the file sgv/src/common/tcscdi/src/IENDataDefine.hpp. Table 2-24.

Table 2-24: IEN Detector Status Codes

DETECTOR STATUS CODE	EXPLANATION
DS_OTHER_NO_ADDITIONAL	Other undefined status, with no additional information
DS_OTHER_ADDITIONAL	Other undefined status, with additional explanatory information (not supported)
DS_FAILED	Detector has failed and the TCS cannot read data from it
DS_OPERATIONAL	Detector is operational
DS_OFF	Detector is offline and not reporting data

2.4.5 Section Events

2.4.5.1 IEN_SECTIONINFO Event Type

The CDI should return the IEN SECTIONINFO event (as described in Table 2-25) if a section is created or any of the member intersections are added to or removed from the section. It should send the value of "-1" in field shortValues[0]if the requested section is deleted from the TCS or does not exist in the TCS.

Table 2-25: Section Configuration Information

FIELD	CONTENTS
eventType	IEN_SECTIONINFO
longValues[0]	ID of the first intersection in the section
longValues[1]	ID of the next intersection in the section
…	
longValues[n]	ID of the last intersection in the section
ShortValues[0]	Section ID, should set to the same value as Event.entityNumber when the requested section exists, or "-1" when the section does not exist in the TCS
octetValues	None

FIELD	CONTENTS
stringValue	None
doubleValue	None

2.4.5.2 IEN_SECTIONSTATE Event Type

The Site Server should return the IEN_SECTIONSTATE event (as described in Table 2-26) to the IEN Site Server when any of the sections defined in the TCS changes its control mode or section timing plan number.

Table 2-26: Section State Data

FIELD	CONTENTS
eventType	IEN_SECTIONSTATE
longValues	None
shortValues[0]	Section control mode, from Table 2-27
shortValues[1]	ID number of timing plan currently set for intersections contained by the section
octetValues	None
stringValue	None
doubleValue	None

The IEN section signal control modes (listed below in Table 2-27) are defined in the file sgv/src/common/tcscdi/src/IENDataDefine.hpp.

Table 2-27: IEN Section Control Modes

SECTION SIGNAL CONTROL MODE	EXPLANATION
SSC_OTHER_NO_ADDITIONAL	Other undefined section control mode, with no additional information
SSC_OTHER_ADDITIONAL	Other undefined section control mode, with additional explanatory information (not supported)
SSC_FREE	Intersections in the section should run free
SSC_FIXED_TIME	Intersections in the section should run fixed time timing plans
SSC_TIME_BASE_COORDINATION	Intersections in the section should run coordinated timing plan
SSC_ACTUATED	Intersections in the section should run fully actuated
SSC_SEMI_ACTUATED	Intersections in the section should run semi-actuated
SSC_CRITICAL_INTERSECTION_CONTROL	Intersections in the section should change the split times in their timing plans based on detector data
SSC_TRAFFIC_RESPONSIVE	Intersections in the section should choose their timing plans based on detector data
SSC_ADAPTIVE	Intersections in the section should define their cycle length, offset, and split times based on detector

SECTION SIGNAL CONTROL MODE	EXPLANATION
	data
SSC_TRANSITION	Not used
SSC_EXTERNAL	Intersections in the section should follow external requests to choose timing plans

3. EXAMPLE IMPLEMENTATION: SERIES 2000 TCS CDI

This section presents the Series 2000 TCS CDI as an example TCS CDI implementation. TCS CDI implementations for other traffic control systems may vary, but they must comply with the TCS CDI interface requirements that were described above.

3.1 CORBA ORBS USED

The IEN Site Server process runs on Microsoft Windows on an Intel Pentium. It uses the TAO ORB, which is freely available at http://www.cs.wustl.edu/~schmidt/corba.html.

The Series 2000 TCS CDI runs on Open-VMS on the HP Alpha Platform. It uses omniORB, because TAO has not been ported to Open-VMS. The version of omniORB that was used implements version 2.2 of the CORBA Specification. The omniORB ORB is also freely available at http://omniorb.sourceforge.net/.

3.2 IMPLEMENTATION ENVIRONMENT

TransCore wrote the Series 2000 TCS CDI in the C++ programming language. It contains conditional compilation to build the actual TCS CDI or a CDI simulator.

3.3 CONFIGURATION DATA

The Series 2000 TCS CDI program reads configuration data from a disk file using a *CfgFile object.* The configuration file contains corridor, site, and system ID data to identify the TCS CDI on the network, and a list of the intersection controllers, system detectors, and sections that the CDI will export to the IEN. A listing of the file is included in *Appendix B: Series 2000 TCS CDI Configuration File*.

3.4 SERIES 2000 TCS CDI MAIN ROUTINE

This section describes the main routine for the Series 2000 TCS CDI. The CDI runs as a single process on the Series 2000 server computer. The code itself can be found in the file sgv/src/common/tcscdi/src/s2kiencdi.cpp.

The initial section of the code gets the site ID and configuration file name from the command line.

```
int main(int argc, char **argv)

{

    const char * pRoutineName = "S2KIENCDI::Main    : ";


    // Check configuration file

    vector<string> optvals;
```

```
    TcOptParser opt(argc, argv);


    string cfgFile = opt.getOpt("config", optvals) ?
                                    *(optvals.begin()) : "TCSData.cfg" ;



#ifdef DEBUG_PRINT
    cout << pRoutineName << "Read from configuration file: " << cfgFile;
    cout << endl << flush;
#endif


    // Setup site id
    long siteid = 0;


    string retval = opt.getOpt("siteid", optvals) ? *(optvals.begin()) : "0" ;
    siteid = atoi( retval.c_str() );



#ifdef DEBUG_PRINT
    cout << pRoutineName << "System linked to site " << siteid << endl;
#endif
```

The next section of code initializes the ORB object.

```
       try
       {
           CORBA::Object_var obj;

           // Initialize ORB
           CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB3");

           if (CORBA::is_nil(orb))
           {
               cerr << "can't initiate ORB" << endl;
               return -1;
           }

   #ifdef DEBUG_PRINT
           cout << pRoutineName << "ORB initialized" << endl << flush;
   #endif
```

The following section of code initializes the Portable Object Adapter that manages the CORBA server objects that the TCS CDI creates.

```
        // Get root POA
        obj = orb->resolve_initial_references("RootPOA");

        PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

        if (CORBA::is_nil(poa))
        {
            cerr << pRoutineName << "Can't resolve RootPOA" << endl;

            return -1;
        }

#ifdef DEBUG_PRINT
        cout << pRoutineName << "Root POA resolved" << endl << flush;
#endif

        // Activate the POA
        PortableServer::POAManager_var poaManager = poa->the_POAManager();
        poaManager->activate();


#ifdef DEBUG_PRINT
        cout << pRoutineName << "Root POA activated" << endl << flush;
#endif
```

The following code obtains a reference to the CORBA naming service.

```
        // Get naming service
        obj = orb->resolve_initial_references("NameService");

        CosNaming::NamingContext_var ncRef =
                                    CosNaming::NamingContext::_narrow(obj);
        if (CORBA::is_nil(ncRef))
        {
            cerr << pRoutineName << "Can't resolve root naming context"<<endl;

            return -1;
        }

#ifdef DEBUG_PRINT
        cout << pRoutineName << "Root naming context resolved" << endl << flush;
#endif
```

The following code creates the CDI's instance of the *TCSDataAF* and *TCSCommandAF*. The Site Server uses these objects to get *TcsDataAccessor* and *TcsCommandAccessor* objects, respectively, for accessing TCS data and sending commands to it.

```
    // Init TCSDataAF servant
    TCSDataAF* daf_i = new TCSDataAF(cfgFile);

    TCSData::DataAccessorFactory_var data_servant = daf_i->_this();

    //init TCSCommandAF servant
    TCSCommandAF* caf_i = new TCSCommandAF(cfgFile);

    TCSCommand::CommandAccessorFactory_var cmd_servant = caf_i->_this();
```

The program then publishes a reference to the *TCSDataAF* object in the naming service. See *Section 2.3: Usage of the CORBA Naming Service*, above for more information on how the names should be published.

```
        // Publish Name with CORBA Naming Service
        char id[20];
        char kind[20];

        CosNaming::Name name;

        sprintf(id, "TCSCDIData%d", siteid);
        sprintf(kind, "Site%d", siteid);

        name.length(1);
        name[0].id = CORBA::string_dup(id);
        name[0].kind = CORBA::string_dup(kind);

        ncRef->rebind(name, data_servant);


#ifdef DEBUG_PRINT
        cout << pRoutineName << "TCSCDI Data Accessor published name: "
                 << id << "/" << kind << endl << flush;
#endif
```

The program publishes a reference to the *TCSCommandAF* object in the naming service.

```
        sprintf(id, "TCSCDICmd%d", siteid);
        sprintf(kind, "Site%d", siteid);

        name.length(1);
        name[0].id = CORBA::string_dup(id);
        name[0].kind = CORBA::string_dup(kind);

        ncRef->rebind(name, cmd_servant);


#ifdef DEBUG_PRINT
        cout << pRoutineName << "TCSCDI Cmnd Accessor published name: "
                                 << id << "/" << kind << endl << flush;
#endif
```

The program then enters the ORB's run loop and stays there indefinitely. In this loop, it will respond to CORBA method invocations from other systems or processes. This implementation is driven only by CORBA method invocations, and does not do background processing using other threads.

```
        // Enter the ORB event loop
        orb->run();
```

The catch clauses and other code handle exceptions thrown by the CORBA ORB code. The program will not execute any of this code in normal conditions.

```
        CORBA::release(orb);

    }   catch(const CORBA::SystemException& se)
        {
            cerr << pRoutineName << "Main() caught CORBA::SystemException: ";
            cerr << se << endl;

            return -1;
        }

        catch(const CORBA::Exception& e)
```

```
        {
            cerr << pRoutineName << "Main() caught CORBA::Exception: ";
            cerr << e << endl;

            return -2;
        }

#ifdef DEBUG_PRINT
        cout << pRoutineName << "S2kcdi server shutdown" << endl;
#endif

        return 0;
```

4. APPENDICES

4.1 APPENDIX A: TCS CDI CORBA IDL FILES

4.1.1 IENRTData.idl

```
//------------------------------------------------------------------------
// Copyright 2001 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
//------------------------------------------------------------------------
// $Id: IENRTData.idl,v 1.16 2004/04/20 18:48:15 build Exp $
//
// CORBA type definitions for IEN Real-time Data Distribution subsystem.
//------------------------------------------------------------------------

#ifndef IENRTDATA_IDL
#define IENRTDATA_IDL

#pragma prefix "transcore.com"
module IENRTData
{
    const string IDLFileID = "$Id: IENRTData.idl,v 1.16 2004/04/20 18:48:15
build Exp $";

    struct Event
    {
        short            entityNumber;  // entity number (unique to system
and event type)
        short            ienEventType;  // type of event (see above)
        long             timeStamp;     // in the format HHMMSS
        sequence<long>   longValues;    // sequence of 32-bit values
        sequence<short>  shortValues;   // sequence of 16-bit values
        sequence<octet>  octetValues;   // sequence of bytes
        string           stringValue;
        double           doubleValue;
    };
    typedef sequence<Event> EventSeq;

    // Event group, consisting of a sequence of events and the identity
    // of the generating system.
    struct EventGroup
    {
        short        corridor;// id for corridor generating this events
        short        site;  // id for site generating this events
        short        system;  // id for system generating this events
        short        sites; // bit map of sites that request these data
        EventSeq     events;  // event data
    };
    typedef sequence<EventGroup> EventGroupSeq;


      // smm/TransCore/atl 2/04 - Added next group of Device Type definitions
      //                          so as to have all defice types defined in a
      //                          central location.

      // IEN System wide device types (Note: When changing number of
```

```
    // items in this list, modify value of DT_COUNT to reflect
    // correct number)
    enum DeviceType
    {
        DT_SYSTEM,
        DT_SCHEDULE,
        DT_INTERSECTION,
        DT_SECTION,
        DT_DETECTOR,
        DT_SIGN,
        DT_CAMERA,
        DT_HAR
    };

    // Constant used to indicate number of type we have defined
    const short DT_COUNT = 8;

};

#endif

//-----------------------------------------------------------------------------
// Copyright 2001 County of Los Angeles. All Rights Reserved.
//-----------------------------------------------------------------------------
```

4.1.2 TCS.idl

```
//-----------------------------------------------------------------------------
// Copyright 2001 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ----------------------------------------------------------------------------
// $Id: tcs.idl,v 1.15 2004/04/20 18:48:15 build Exp $
//
// Basic definitions for CORBA interface to Generic TCS
// ----------------------------------------------------------------------------

#ifndef TCS_IDL
#define TCS_IDL

#pragma prefix "transcore.com"


#include "IENRTData.idl"


module TCS
{
    typedef sequence<IENRTData::DeviceType> DeviceTypeList;

    /// Modes of operation
    enum Mode
    {
      NORMAL,
    LOCAL_TOD,
      FREE,
      TOD,
    RESPONSIVE,
      MANUAL,
      RELEASE
    };

    /// Functions that can be activated or deactivated on
    /// a per-device basis
    enum SpecialFunction
    {
        FLASH,
        SPECIAL_FUNCTION_1,
        SPECIAL_FUNCTION_2,
        SPECIAL_FUNCTION_3,
        SPECIAL_FUNCTION_4
    };

    /// Device identifier number
    typedef short DeviceID;

    /// Unique identifier for a device in a given TCS
    struct Device
    {
        IENRTData::DeviceType type;
        DeviceID   id;
    };
```

```
/// List of Device elements
typedef sequence<Device> DeviceList;

/// This exception is thrown when something goes wrong that
/// is not covered by a more specific exception
exception Error
{
    string reason;
};

/// Thrown by methods if devices are specified that
/// are unknown to TCS or not fully configured
exception UnknownDevices
{
    DeviceList unknowns;
};

/// Version number (major.minor.revision)
struct Version
{
    short major;
    short minor;
    short revision;
};

/// System status codes
enum Status
{
    /// Running normally
    SYSTEM_NORMAL,

    /// Initializing; features may be unavailable or uninitialized
    SYSTEM_STARTING,

    /// Shutting down; features may be unavailable or no longer updated
    SYSTEM_STOPPING,

    /// TCS is not running
    SYSTEM_SHUTDOWN,

    /// TCS *is unable to run
    SYSTEM_ERROR
};

/// This exception is thrown when a client attempts  an operation while
/// TCS is in a state that does not allow it.
exception SystemStatusException
{
    Status systemStatus;
};

/// Interface that provides client with means to discover what
/// devices are available from the TCS, and other
/// high-level aspects of the system.
interface ConfigurationAccessor
{
    /// @return Version number for TCS CORBA interface
```

```
        readonly attribute Version interfaceVersion;

        /// @return Version of TCS software
        readonly attribute Version systemVersion;

        /// @return Name of TCS system
        readonly attribute string systemName;

        /// @return Current status of TCS
        readonly attribute Status systemStatus;

        /// @return list of configured devices of the given types
        DeviceList getAvailableDevices(in DeviceTypeList types);
    };
};

#endif

//-----------------------------------------------------------------------------
// Copyright 2001, 2004 County of Los Angeles. All Rights Reserved.
//-----------------------------------------------------------------------------
```

4.1.3 TCSData.idl

```
// ------------------------------------------------------------------------
// Copyright 2001, 2004 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ------------------------------------------------------------------------
// $Id: tcsdata.idl,v 1.14 2004/05/04 15:42:12 build Exp $
//
// CORBA interface to TCS data.
// ------------------------------------------------------------------------

#ifndef TCSDATA_IDL
#define TCSDATA_IDL

#include "TCS.idl"

#pragma prefix "transcore.com"

module TCSData
{
    /// Major version, minor version, and revision of this IDL interface
    const short majorVersion = 2;
    const short minorVersion = 0;
    const short revision     = 1;

    /// Item codes are not defined in IDL; they are
    /// documented elsewhere
    typedef short Code;

    /// List of data item code elements
    typedef sequence<Code> CodeList;

    /// Data types returned for a given device type

    struct DeviceDataTypes
    {
        IENRTData::DeviceType type;
        CodeList dataTypes;
    };

    /// List of data types for all supported devices.
    typedef sequence<DeviceDataTypes> DeviceDataTypeList;

    /// A device ID followed by a list of data codes supported by the
    /// device, and a flag to indicate if the CDI should retrieve all
    /// data for the device or only changed data.

    struct DeviceCode
    {
        TCS::Device device;
        CodeList    dataCodes;
        boolean     changedOnly;
    };

    /// List of DeviceCode elements
    typedef sequence<DeviceCode> DeviceCodeList;
```

```
    /// Interface for retrieving data from TCS
    interface DataAccessor: TCS::ConfigurationAccessor
    {
        /// Instance name passed to DataAccessorFactory's
        /// CreateDataAccessor() method to create this instance.
        readonly attribute string clientName;

        /// Client calls this method when finished with this
        ///  DataAccessor.  Releases all resources associated with this
        /// instance.
        void destroy();

        /// @return the configured device list for this instance.
        TCS::DeviceList getDeviceList();

        /// Get the data type codes supported for all device types
        /// for which this CDI returns data.
        /// @return       supported data types for all supported devices
        DeviceDataTypeList deviceDataTypes( );

        /// @return Data items for input device list.
        ///
        /// @param devices  List of devices for which to get data.  Each
        ///                 entry in the list has a device ID, requested
        ///                 data types, and a changeOnly flag indicating
        ///                 if the method should retrieve only changed
        ///                 data (if true), or all known data (if false).
        /// @return         Sequence of IEN events containing requested
        ///                 requested data
        /// @throws         SystemStatusException if system not currently
        ///                 running
        /// @throws         Error if a device ID or data type in the
        ///                 device list is not supported.
        IENRTData::EventSeq getDeviceEventDataList(
                            in DeviceCodeList devices)
                            raises (TCS::SystemStatusException,
                                    TCS::Error);
};


    /// Interface for creating instances of DataAccessor
    interface DataAccessorFactory
    {
        /// Create an instance of DataAccessor.
        ///
        /// @param clientName  Text identifying the user
        ///                    of this interface.  For
        ///                    informational and diagnostic
        ///                    purposes only.
        ///
        /// @param option      Provides access to special
        ///                    functionality (generally for
        ///                    debugging or testing purposes).
        ///                    Always pass 0 unless you know
        ///                    what you're doing.
        /// @throws Error      If the client name is empty or the
```

```
        ///                     option is not supported

        DataAccessor createDataAccessor(    in string clientName,
                                            in long   option )
                                            raises ( TCS::Error );
    };
};

#endif

//---------------------------------------------------------------------------
// Copyright 2001, 2004 County of Los Angeles. All Rights Reserved.
//---------------------------------------------------------------------------
```

4.1.4 TCSCommand.idl

```
//-----------------------------------------------------------------------------
// Copyright 2001 County of Los Angeles. All Rights Reserved.
//
// Developed by TransCore
// ----------------------------------------------------------------------------
// $Id: tcscommand.idl,v 1.15 2004/04/20 18:48:15 build Exp $
//
// CORBA interface to TCS commands.
// ----------------------------------------------------------------------------

#ifndef TCSCOMMAND_IDL
#define TCSCOMMAND_IDL

#include "tcs.idl"

#pragma prefix "transcore.com"
module TCSCommand
{
    /// Major version, minor version, and revision of this IDL interface
    const short majorVersion = 2;
    const short minorVersion = 0;
    const short revision     = 1;

    // Thrown when a request is made to activate/deactivate
    // a special function that is not supported by one or
    // more of the devices specified in the request.
    exception UnsuppportedSpecialFunction
    {
        TCS::SpecialFunction function;
        TCS::DeviceList      devices;
    };

    /// Thrown when a plan number is specified that is not
    /// supported by one or more of the devices in the
    /// command.
    exception InvalidPlanNumber
    {
        short                planNumber;
        TCS::DeviceList devices;
    };

    /// Thrown when a mode is specified that is not supported
    /// by one or more of the devices in the request.
    exception InvalidMode
    {
        TCS::Mode       invMode;
        TCS::DeviceList devices;
    };

    /// Thrown when a special function is not supported
    /// by one or more of the devices in the request.
    exception UnsupportedSpecialFunction
    {
        TCS::SpecialFunction function;
        TCS::DeviceList      devices;
```

```
    };

    /// Thrown when the TCS interface is not accepting commands
    /// (because it has been manually disabled)
    exception CommandsNotAccepted
    {
        string reason;
    };

    /// Interface that allows clients to send commands to TCS
    interface CommandAccessor: TCS::ConfigurationAccessor
    {
        /// Text passed to DataAccessorFactory::CreateDataAccessor()
        /// to create this instance.
        readonly attribute string clientName;

        /// Client calls this method when finished with this CommandAcceptor.
        /// Releases all resources associated with this instance.
        void destroy();

            /// Change CDI plan number
        ///
        /// @param devices    List of devices for which to change the plan
        /// @param planNumber New plan number for the requested devices
        void setCDIPlan(
            in TCS::DeviceList devices,
            in short           planNumber
        ) raises (
            CommandsNotAccepted,
            InvalidPlanNumber,
            TCS::UnknownDevices,
            TCS::SystemStatusException,
            TCS::Error
        );

        /// Change operational mode of specified devices.
        ///
        /// @param devices    List of devices for which to change the mode
        /// @param newMode    New operational mode for the devices on the
list
        void changeMode(
            in TCS::DeviceList devices,
            in TCS::Mode       newMode
        ) raises (
            CommandsNotAccepted,
            InvalidMode,
            TCS::UnknownDevices,
            TCS::SystemStatusException,
            TCS::Error
        );

        /// Release IEN control of a list of TCS devices
        ///
        /// @param devices    List of devices for which to release IEN
control
        void releaseControl(
            in TCS::DeviceList devices
        ) raises (
```

```
            TCS::UnknownDevices,
            TCS::SystemStatusException,
            TCS::Error
        );

    };

    /// Interface for creating instances of CommandAcceptor
    interface CommandAccessorFactory
    {
        /// Create an instance of DataAccessor.
        ///
        /// @param clientName  Text identifying the user
        ///                    of this interface.  For
        ///                    informational and diagnostic
        ///                    purposes.
        ///
        /// @param option      Provides access to special
        ///                    functionality (generally for
        ///                    debugging or testing purposes).
        ///                    Always pass 0 unless you know
        ///                    what you're doing.
        CommandAccessor createCommandAccessor(
            in string clientName,
            in long   option
        ) raises (TCS::Error);
    };
};

#endif
//-----------------------------------------------------------------------------
// Copyright 2001, 2004 County of Los Angeles. All Rights Reserved.
//-----------------------------------------------------------------------------
```

4.2 APPENDIX B: SERIES 2000 TCS CDI CONFIGURATION FILE

The configuration file for the Series 2000 TCS CDI is listed below. The file is located in the source tree at sgv\src\common\tcscdi\vms\tcsdata.cfg.

```
[TCS]

#TCS identifiers
Corridor ID = 1
Site ID = 2
System ID = 1
SystemName = PASS2K

#CDI options
CommandsEnabled = 1
DiagLevel = 1

[Device]

#Pasadena Series 2000 TCS devices
Intersection = 1-999
Detector = 1-2999, 6251-6258
Section = 1-100

[DetectorConfiguration]

# DetectorConfiguration data is not available in Series 2000.
# Config data must be specified below in the following format:
#
# detector id =
#   detClass,
#   detType,
#   direction,
#   laneNumber,
#   roadwayName
#
# Example: 2201 = 3, 2, 2, 1, Colorado Blvd.
#
# See the IENTCSData.idl for detClass and detType enumerations.
# See the location.idl for the direction enumeration.
#
# Default values are specified under detector id "0". Default
# values are reported for detectors that are not listed below.
#
# Note: The CDI must be restarted for changes to be applied.
#

0 = 3, 2, 10, 0, Unknown
```

4.3 APPENDIX C: DIAGNOSTIC SETTINGS IN THE SITE SERVER PROGRAM

The Site Server program can be configured to write diagnostic output to its standard output, which CDI Developers may find helpful when diagnosing problems with CDI programs. The configuration file for the Site Server (usually sitesvr.cfg) contains a "System" section with settings similar to the following:

```
[System]

# system ids
CorridorID = 1
SiteID = 1
SystemID = 1
DiagLevel = 10
```

The DiagLevel configuration item governs diagnostic output that the Site Server prints for data that it receives from a CDI. To change the quantity of CDI communication that the Site Server prints out, change the DiagLevel setting in the configuration file, then stop and restart the Site Server.

Table 4-1 below lists the diagnostic levels available with the Site Server:

Table 4-1: Diagnostic Levels for the Site Server

DIAGNOSTIC LEVEL	EXPLANATION
0	Generate no diagnostic messages for CDI communication
> 0	Print the time required for the Site Server to call the CDI's getDeviceEventDataList method, in milliseconds
10	Print data in intersection related events received from the CDI, as well as the time required to make the getDeviceEventDataList call to the CDI
11	Print data in detector related events received from the CDI, as well as the time required to make the getDeviceEventDataList call to the CDI
12	Print data in section related events received from the CDI, as well as the time required to make the getDeviceEventDataList call to the CDI
>20	At startup, prints a summary of the types of device data that the Site Server is requesting from the TCS CDI, as well as the time required to make the getDeviceEventDataList call to the CDI
>22	At startup, prints a summary of the types of device data that the Site Server is requesting from the TCS CDI. Each time that the Site Server requests calls the CDI's getDeviceEventDataList method, prints the list of devices and data types requested, as well as the time required to make the call.